HAL
open science

Contributions to Building Efficient and Robust State-Machine Replication Protocols

Vivien Quéma

To cite this version:

Vivien Quéma. Contributions to Building Efficient and Robust State-Machine Replication Protocols. Réseaux et télécommunications [cs.NI]. Université de Grenoble, 2010. tel-00540897

HAL Id: tel-00540897

https://theses.hal.science/tel-00540897v1

Submitted on 29 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GRENOBLE INP

Rapport scientifique présenté pour l'obtention d'une

Habilitation à Diriger des Recherches

Spécialité : " Informatique"

soutenue publiquement par

VIVIEN QUÉMA

le

Titre :

Contributions to Building Efficient and Robust State-Machine Replication Protocols

---

JURY

Pr. Yves Denneulin	Président
Pr. Kenneth Birman	Rapporteur
Pr. Paulo Verissimo	Rapporteur
Pr. Willy Zwaenepoel	Rapporteur
Pr. Rachid Guerraoui	Examinateur
Dr. Anne-Marie Kermarrec	Examinateur
Dr. Gilles Muller	Examinateur

Abstract

State machine replication (SMR) is a software technique for tolerating failures using commodity hardware. The critical service to be made fault-tolerant is modeled by a state machine. Several, possibly different, copies of the state machine are then deployed on different nodes. Clients of the service access the replicas through a SMR protocol which ensures that, despite concurrency and failures, replicas perform client requests in the same order.

Two objectives underly the design and implementation of a SMR protocol: *robustness* and *performance*. Robustness conveys the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. On the other hand, performance measures the time it takes to respond to a request (latency) and the number of requests that can be processed per time unit (throughput).

In this thesis, we present two contributions to state machine replication. The first contribution is LCR, a uniform total order broadcast (UTO-broadcast) protocol that is throughput optimal in failure-free periods. LCR can be used to totally order the requests received by a replicated state machine. LCR has been designed for small clusters of homogeneous machines interconnected by a local area network. It relies on a *perfect* failure detector and tolerates the crash failures of all but one replicas. It is based on a ring topology and only relies on point-to-point inter-process communication. We benchmark an implementation of LCR against two of the most widely used group communication packages and show that LCR provides higher throughput than them, over a large number of setups.

The second contribution is Abstract, a new abstraction to simplify the design, proof and implementation of SMR protocols. Abstract focuses on the most robust class of SMR protocols, i.e. those tolerating arbitrary (client and replica) failures. Such protocols are called Byzantine Fault Tolerant (BFT) protocols. We treat a BFT protocol as a composition of instances of our abstraction. Each instance is developed and analyzed independently. To illustrate our approach, we first show how, with our abstraction, the benefits of a BFT protocol like Zyzzyva could have been developed using less than 24% of the actual code of Zyzzyva. We then present *Aliph*, a new BFT protocol that outperforms previous BFT protocols both in terms of latency (by up to 30%) and throughput (by up to 360%).

Acknowledgments

I would first like to thank Yves Denneulin for presiding over my thesis committee. I would also like to thank Ken Birman, Paulo Verissimo, and Willy Zwaenepoel who accepted to report on my work. Many thanks also to Rachid Guerraoui, Anne-Marie Kermarrec, and Gilles Muller who accepted to be part of my thesis committee. It's a great honor for me.

The work presented in this thesis would not have been achievable without the invaluable help of Rachid Guerraoui. He deserves special thanks for all the things he has done for me for many years. It is both an honor and a great pleasure for me to work with him.

I would also like to thank Ron Levy with whom I designed the LCR protocol, and Marko Vukolić with whom I designed Abstract. I am also grateful to Nikola Knežević who helped implementing and evaluating Abstract.

I am also very grateful to all my colleagues at CNRS, INRIA, and University of Grenoble, and particularly to Brigitte Plateau and Jean-Bernard Stefani for their constant support for many years.

Finally, I would also like to thank all my senior colleagues and mentors who help me learn how to do research: Lorenzo Alvisi, Roberto Baldoni, Mike Dahlin, Rachid Guerraoui, Anne-Marie Kermarrec, and Gilles Muller.

Contents

Abstract **3**

Acknowledgments **5**

Contents **5**

1 Introduction **9**
1.1 Context 10
1.2 Contributions 10
1.3 Outline 12

2 LCR: a Throughput Optimal Total Order Broadcast Protocol **13**
2.1 Introduction 13
2.2 Model 15
2.2.1 System Model 15
2.2.2 Performance Model 15
2.2.3 Throughput 15
2.3 Related Work 16
2.3.1 Fixed Sequencer 16
2.3.2 Moving Sequencer 16
2.3.3 Privilege-based Protocols 17
2.3.4 Communication History-based Protocols 17
2.3.5 Destination Agreement 18
2.4 The LCR Protocol 18
2.4.1 Total Order Definition 18
2.4.2 Failure-free Behavior 19
2.4.3 Group Membership Changes 22
2.4.4 Correctness 22
2.5 Theoretical Analysis 25
2.5.1 Throughput 25
2.5.2 Fairness 26
2.5.3 Latency 27
2.6 Experimental Evaluation 27
2.6.1 Experimental Setup 27
2.6.2 Throughput 28
2.6.3 Response Time 32
2.6.4 Fairness 34
2.6.5 CPU Usage 34
2.7 Conclusion 35

3 ABsTRACT: a Modular Approach to Building BFT Protocols **39**
3.1 Introduction . . . 39
3.2 Abstract . . . 41
3.2.1 Switching . . . 41
3.2.2 Illustration . . . 43
3.2.3 A View-Change Perspective . . . 43
3.2.4 Misbehaving Clients . . . 44
3.3 System Model . . . 44
3.4 Putting Abstract to Work: AZyzzyva . . . 44
3.4.1 Protocol Overview . . . 45
3.4.2 ZLight . . . 45
3.4.3 Backup . . . 46
3.4.4 Qualitative Assessment . . . 46
3.4.5 Performance Evaluation . . . 47
3.5 Putting Abstract to Really Work: Aliph . . . 48
3.5.1 Protocol Overview . . . 48
3.5.2 Evaluation . . . 52
3.6 Related Work . . . 59
3.7 Conclusion . . . 61

4 Conclusion **63**
4.1 Summary of Contributions . . . 63
4.2 Future Work . . . 64

Bibliography **66**

A Abstract**: specification** **75**

B Abstract**: detailed protocol descriptions** **77**
B.1 Notations . . . 77
B.2 ZLight Details . . . 78
B.3 Quorum Details . . . 78
B.4 Chain Details . . . 79
B.5 Panicking Mechanism . . . 80
B.6 Handling Init Histories and Switching . . . 80
B.6.1 Lightweight Checkpointing Subprotocol . . . 81

C Abstract**: correctness proofs** **83**
C.1 ZLight and Quorum . . . 83
C.2 Chain . . . 85

Chapter 1

Introduction

Since I obtained my PhD in December 2005, I have been working on four research topics: component-based software engineering, large-scale distributed systems, system support for multicore architectures, and replication-based fault tolerance. I briefly describe below the main results achieved for each topic.

- **Component-based software engineering.** I have contributed to the design of the Fractal component model. This component model is original in the sense that it was the first model allowing component sharing (a component can simultaneously be a sub-component of two distinct components). Moreover, another distinguishing feature in this model is that components can be endowed with arbitrary reflective capabilities, from plain black-box objects to components that allow a fine-grained manipulation of their internal structure. This work has been published in SP&E in 2006 [24]. We also designed an extensible toolset for easing the development of component-based software systems. The originality of this toolset is that it does not define its own input Architecture Description Language (ADL), but rather uses a grammar description mechanism to accept various input languages, e.g. ADLs, Interface Definition Languages (IDLs), Domain Specific Languages (DSLs). This work has been published at ICSE 2007 [72].

- **Large-scale distributed systems.** I have participated in the design of several gossip-based protocols. Such protocols are used in very large-scale peer-to-peer systems, where nodes cannot have a full knowledge about other nodes in the network. In gossip-based protocols, each node knows a (constantly changing) subset of other nodes in the network, with which it periodically exchanges data. We designed a replication protocol that ensures eventual consistency in large-scale distributed systems subject to network partitions and asynchrony. This work has been published at SSS 2006 [13]. We also designed TERA, a topic-based publish-subscribe protocol. This work has been published at DEBS 2007 [12]. We proposed $\mathcal{N}ylon$, the first gossip-based protocol able to work despite the presence of Network Address Translation (NAT) gateways. This work has been published at ICDCS 2009 [65]. We were also among the first to design and deploy a gossip-based streaming protocol able to take into account the heterogeneity in node capabilities. This protocol, called HEAP, has been published at DSN 2009 [50] and Middleware 2009 [49]. Finally, we have designed FireSpam, the first spam-resilient gossip-based broadcasting protocol. This work has been published at SRDS 2010 [80].

- **System support for multicore architectures.** I have participated in the design of a new runtime supporting event-driven programming for multicore platforms. Event-

driven programming is a popular approach for the development of efficient applications such as networked systems. This programming and execution model is based on *continuation-passing* between short-lived and *cooperatively-scheduled* tasks. The originality of our runtime is that it encompasses an efficient workstealing algorithm in charge of balancing the execution of very fine-grain tasks on the available cores. This work has been published at ICDCS 2010 [55].

- **Replication-based fault tolerance.** I have participated in the design of algorithms and abstractions that are used as core building blocks in reliable distributed systems. In particular, we designed LCR, the first total order broadcast protocol that is throughput-optimal in LAN environments. This work has been published at DSN 2006 [60] and in ACM TOCS in 2010 [61]. We have also designed a throughput-efficient atomic storage algorithm that has been published at ICDCS 2007 [59]. This protocol, as well as LCR, achieve very high throughput by organizing nodes in a ring topology and by only relying on point-to-point communications. Finally, we have designed a new abstraction to ease the design, proof and implementation of Byzantine-resilient replication protocols. The originality of our abstraction lies in the fact that it is possible to build Byzantine-resilient replication protocols as a composition of distinct instances of our abtraction. Each instance can be developed and analyzed independently. This work has been published at EuroSys 2010 [58] and was awarded best paper of the conference.

In this document, I focus on the work I carried out on replication-based fault tolerance. In the reminder of this section, I first describe the context of this work. Then I briefly introduce the main contributions. Finally, I describe the outline of the document.

1.1 Context

As an ever increasing number of critical tasks are being delegated to computers, the unforeseen failure of a computer can have catastrophic consequences. Unfortunately, the observed increase of computing power as predicted by Moore's law has not been coupled with a similar increase in reliability. On the other hand, because of rapidly decreasing hardware costs, ensuring fault tolerance through state machine replication [86] is gaining in popularity.

State machine replication is a software technique for tolerating failures using commodity hardware. It has been initially proposed by Lamport in his seminal paper "*Time, Clocks, and the Ordering of Events in a Distributed System*" [69]. The critical service to be made fault-tolerant is modeled by a state machine. Several, possibly different, copies of the state machine are then deployed on different nodes. Clients of the service access the replicas[1] through a state machine replication protocol which ensures that, despite concurrency and failures, replicas perform client requests in the same order. In a replicated database for instance, all replicas must perform the same write queries (INSERT and UPDATE) in the same order. Read queries (SELECT) do not change the state of the replicated database and do not have to be performed by all replicas.

Two objectives underly the design and implementation of a state machine replication protocol: *robustness* and *performance*. Robustness conveys the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. On the other hand, performance measures the time it takes to respond to a request (latency) and the number of requests that can be processed per time unit (throughput).

[1] Replicas are also called "processes" throughout the thesis.

1.2 Contributions

In this thesis, two contributions to state machine replication are presented. I briefly introduce them below before presenting them in details in the next chapters.

The LCR Uniform Total Order Broadcast Protocol. The first contribution presented in this thesis is LCR [60, 61, 73], a uniform total order broadcast (UTO-broadcast) protocol that is throughput optimal in failure-free periods. A UTO-broadcast protocol is a primitive that can be used by replicas of a replicated state machine to order requests received from clients. More formally, a UTO-broadcast protocol ensures the following for all messages that are broadcast [62]: (1) *Uniform agreement*: if a replica delivers a message m, then all correct replicas eventually deliver m; (2) *Strong uniform total order*: if some replica delivers some message m before message m', then a replica delivers m' only after it has delivered m.

LCR has been designed for small clusters of homogeneous machines interconnected by a local area network. It relies on a *perfect* failure detector (P) [28] and tolerates the crash failures of all but one replicas.

The motivation that led us to develop LCR is that most of the total order broadcast protocols that have been designed in the last two decades [64, 8, 26, 54, 21, 97] target low latency. On the other hand, very few protocols have been designed for high throughput. Nevertheless, we believe that in some high load environments, e.g. database replication for e-commerce, throughput is often as important, if not more, than latency. Indeed, under high load, the time spent by a message in a queue before being actually disseminated can grow indefinitely. A high throughput broadcast protocol reduces this waiting time.

The key to achieving high throughput in LCR is to organize processes in a ring topology: each process always sends messages to the same process, thus avoiding any possible collisions. To eliminate bottlenecks, messages in LCR are sequenced using logical vector clocks instead of a dedicated sequencer. Moreover, using a ring topology, acknowledgement messages can easily be piggy-backed, which further reduces the message overhead.

We have proposed a new (simple) distributed system model that we use to define an upper bound on the throughput that can be achieved by UTO-broadcast protocols. Using this model, we have shown that LCR theoretically matches this lower bound. This model is a round-based model [76]: time is decomposed in rounds of equal duration. It assumes that a process can send a message to one (unicast) or more processes (multicast) at the start of each round, and can receive one message sent by other processes at the end of the round[2]. This model is similar to the Postal model [16] with the addition of a multicast primitive which is available on all current local area networks.

We have implemented LCR in order to (i) confirm the theoretical analysis made using our new distributed system model, and (2) compare its performance with that achieved by the two most popular group communication frameworks: Spread [3] and JGroups [14]. Our experiments show that the proposed distributed system model is accurate: the throughput achieved by LCR is very close to the expected one. Our experiments also show that LCR consistently outperforms Spread and JGroups. For instance, with 4 processes, LCR achieves throughput of up to 50% higher than that of Spread and up to 28% higher than that of JGroups.

The Abstract framework. The second contribution presented in this thesis is Abstract [58, 95], a new abstraction that significantly reduces the development cost of state machine replication protocols and makes it significantly easier to develop new efficient ones. Abstract focuses on state machine replication protocols tolerating arbitrary (client and replica) failures. These protocols are commonly called "BFT protocols" (for Byzantine Fault Tolerance).

[2] In contrast with the classical round-based model which assumes that a process can receive several messages at the end of each round.

The motivation that led us to develop Abstract is twofold:

- *BFT protocols are notoriously complex to design and implement.* As an example, the only complete proof of a BFT protocol we know of is that of PBFT [27] and it involves 35 pages. Moreover, any change to an existing protocol, although algorithmically intuitive, is very painful to design and implement (as exemplified by Zyzzyva which can be seen as an extension of PBFT [67]). Regarding implementations, existing research prototypes, although providing often incomplete implementations, involve more than 20.000 lines of (non-trivial) C++ code, e.g., PBFT [27] and Zyzzyva [67].

- *There is no "one size that fits all" BFT protocol.* A recent study [87] of three existing BFT protocols (PBFT [27], Zyzzyva [67], and Q/U [1]) showed that their performance can be heavily impacted by different factors. Our own experiences indeed reveal that the performance of BFT protocols are impacted by the size of messages, the actual number of clients, the total number of replicas, as well as the cost of the cryptographic primitives being used. There are thus good reasons to believe that new BFT protocols will need to be designed in the future in order to efficiently deal with some specific (possibly new) operating conditions.

In order to address the need to develop new BFT protocols, and to decrease the complexity of this task, we have proposed Abstract. Using Abstract, a BFT protocol is viewed as a composition of instances of our abstraction, with each instance targeting and optimized for specific operating conditions. An instance of Abstract looks like a traditional BFT state machine replication protocol, with one exception: it may sometimes *abort* a client's request. The (non-triviality) condition under which Abstract cannot abort is a generic parameter. At one extreme, one can for example specify a (useless) Abstract instance that could abort in every execution. At the other extreme, one can prevent Abstract from ever aborting: in such a case, Abstract implements a standard BFT state machine replication protocol. Interesting Abstract instances are "weaker" ones, in which an abort is allowed, e.g., if there are failures or asynchrony (or even contention). When such an instance aborts a client request, it returns an unforgeable request history that can be used by the client to "recover" using another instance of Abstract. Any composition of Abstract instances is possible; we expect many of these to lead to interesting flexible BFT protocols.

We illustrate the benefits of Abstract by developing two BFT protocols. The first protocol, *AZyzzyva*, illustrates the ability of Abstract to significantly ease the development of BFT protocols. *AZyzzyva* is a composition of two Abstract instances. It mimics Zyzzyva [67] when there are no asynchrony or failures and requires about $1/4$ of the code of Zyzzyva. The second protocol is *Aliph*, a protocol that demonstrates the ability of Abstract to develop novel efficient BFT protocols. *Aliph* uses three Abstract instances. It achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art BFT protocols.

1.3 Outline

The thesis is organized as follows. Chapter 2 presents the LCR protocol. The Abstract framework is presented in Chapter 3. Chapter 4 concludes the thesis.

Chapter 2

LCR: a Throughput Optimal Total Order Broadcast Protocol

This chapter presents LCR [60, 61, 73], the first throughput optimal uniform total order broadcast protocol. LCR is based on a ring topology. It only relies on point-to-point inter-process communication and has a linear latency with respect to the number of processes. LCR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes. We benchmark an implementation of LCR against Spread [3] and JGroups [14], two of the most widely used group communication packages. We show that LCR provides higher throughput than them, over a large number of setups.

2.1 Introduction

As explained in the Introduction of this thesis, uniform total order broadcast (UTO-broadcast) protocols can be used to implement state machine replication [86]. The role of the UTO-broadcast protocol is to ensure that all replicas perform the same operations on their copy in the same order, even if they subsequently fail. UTO-broadcast can be formally defined as follows [62]: (1) *Uniform agreement*: if a replica delivers a message m, then all correct replicas eventually deliver m; (2) *Strong uniform total order*: if some replica delivers some message m before message m', then a replica delivers m' only after it has delivered m.

Clearly, even though UTO-broadcast is very useful, not all applications need the strong guarantees that it provides. Some applications might only need reliable or even best-effort broadcast. We will show however that there are no weaker broadcast protocols that can obtain higher throughput than the protocol described in this chapter. In a sense, the strong uniform total order guarantees provided by our protocol are free.

Latency vs. Throughput. Historically, most total order protocols have been devised for low broadcast latency [64, 8, 26, 54, 21, 97]. Latency usually measures the time required to complete a single message broadcast without contention. (It is typically captured by a number of rounds in the classical model of [76]: in that model, at the start of each round a process can send a message to one or more processes. It receives the messages sent by other processes at the end of the round.) On the other hand, very few protocols have been designed for high throughput. Throughput measures the number of broadcasts that the processes can complete per time unit. In some high load environments, e.g. database replication for e-commerce, throughput is often as important, if not more, than latency. Indeed, under high load, the time spent by a message in a queue before being actually disseminated can grow indefinitely. A high throughput broadcast protocol reduces this waiting time.

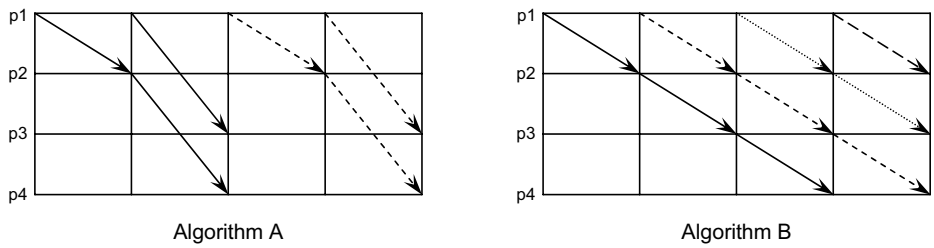


Figure 2.1: Latency vs. Throughput: comparison of two broadcast algorithms A and B. Process p_1 initiates the broadcast. Broadcast latency is 2 rounds for A and 3 rounds for B. However, B has higher throughput than A: in algorithm B one broadcast is completed every round, while in algorithm A only one broadcast is completed every 2 rounds.

Maybe not surprisingly, protocols that achieve low latency often fail to provide high throughput. To illustrate this, consider the example depicted in Figure 2.1:

1. In algorithm $\mathcal{A}$, process p_1 first sends a message to p_2. In the next step p_2 forwards the message to p_4 and at the same time p_1 sends the message to p_3.

2. In algorithm $\mathcal{B}$, process p_1 first sends a message to p_2. In the next step p_2 forwards the message to p_3 and finally p_3 forwards the message to p_4.

Algorithm $\mathcal{A}$ has a latency of 2 whereas algorithm $\mathcal{B}$ a latency of 3. Latency wise, $\mathcal{A}$ is better than $\mathcal{B}$. If we look at the throughput however, we see that $\mathcal{A}$ can only start a new broadcast every 2 time units, while $\mathcal{B}$ can broadcast a new message every time unit. Therefore even though the latency of $\mathcal{B}$ is higher than that of $\mathcal{A}$, the maximal throughput of $\mathcal{B}$ is twice that of $\mathcal{A}$.

Contributions. In this chapter we present LCR [60, 61, 73], a uniform total order broadcast protocol that is throughput optimal in failure-free periods. LCR relies on point-to-point communication channels between processes. It uses logical clocks and a ring topology (hence the name). The ring topology ensures that each process always sends messages to the same process, thus avoiding any possible collisions. To eliminate bottlenecks, messages in LCR are sequenced using logical vector clocks instead of a dedicated sequencer. Furthermore, the ring topology allows all acknowledgement messages to be piggy-backed, reducing the message overhead. These two characteristics ensure throughput optimality and fairness, regardless of the type of traffic. In our context, fairness conveys the equal opportunity of processes to have their broadcast messages delivered.

We give a full analysis of LCR's performance and fairness. We also report on performance results based on a C implementation of LCR that relies on TCP channels. The implementations are benchmarked against Spread and JGroups on a cluster of 9 machines and we show that LCR consistently delivers the highest throughput. For instance, with 4 machines, LCR achieves throughput of up to 50% higher than that of Spread and up to 28% higher than that of JGroups.

Roadmap. Section 2.2 introduces the relevant system and performance models. Section 2.3 gives an overview of the related work. We describe LCR in Section 2.4, then we give an analytical evaluation of it in Section 2.5, and we report on our experimental evaluation in Section 2.6. Section 2.7 concludes the chapter with some final remarks.

2.2 Model

2.2.1 System Model

Our context is a small cluster of homogeneous machines interconnected by a local area network. In our protocol, each of the n machines (or processes) creates a TCP connection to only a single process and maintains this connection during the entire execution of the protocol (unless the process fails). Because of the simple communication pattern, the homogeneous environment, and low local area network latency, it is reasonable to assume that, when a TCP connection fails, the server on the other side of the connection failed [46]. We thus directly implement the abstraction of a *perfect* failure detector (P) [28] to which each process has access.

2.2.2 Performance Model

Analyzing the performance of a communication abstraction requires a precise distributed system model. Some models only address point-to-point networks, where no native broadcast primitive is available [41, 15]. The model of [91], which was recently proposed to evaluate total order broadcast protocols, assumes that a process cannot simultaneously send and receive a message. This does clearly not capture modern network cards for these provide full duplex connectivity. Round-based models [76] are in that sense more convenient as they assume that a process can send a message to one or more processes at the start of each round, and can receive the messages sent by other processes at the end of the round. Whereas this model is well-suited for proving lower bounds on the latency of protocols, it is however not appropriate for making realistic predictions about the throughput. In particular, it is not realistic to consider that several messages can be simultaneously received by the same process.

In this chapter, we analyze protocols using the model used in [60, 59]. This model assumes that processes can send one message to one (unicast) or more processes (multicast) at the start of each round, but can only receive a single message sent by other processes at the end of the round. If more than one message is sent to the same process in the same round, these messages will be received in different rounds. The rationale behind this model is that machines in a cluster are each connected to a switch via a twisted pair ethernet cable. As modern network cards are full-duplex, each machine can simultaneously send and receive messages. Moreover, as the same physical cable is used to send and receive messages, it makes sense to make the very same assumption on the number of messages that can be received and on the number of messages that can be sent in one round. The model we use can thus be described as follows: in each round k, every process p_i can execute the following steps:

1. p_i computes the message for round k, $m(i, k)$,
2. p_i sends $m(i, k)$ to all or a subset of processes,
3. p_i receives at most one message sent at round k.

In a sense, our model is similar to the Postal model [16] with the addition of a multicast primitive which is available on all current local area networks.

2.2.3 Throughput

Basically, throughput captures the average number of completed broadcasts per round. A complete broadcast of message m means that all processes delivered m. In this model, a broadcast protocol is optimal if it achieves an average of one complete broadcast per round regardless of the number of broadcasters. Considering a cluster with n processes, we seek for

an optimal throughput with k simultaneous broadcasters, k ranging from 1 to n. When evaluating throughput, we assume a variable number k of processes continuously sending messages, where $1 \leq k \leq n$. Using this assumption, we can accurately model bursty broadcast patterns. In the general case with random broadcast patterns, we can observe the following: as soon as the load on the broadcast system approaches the maximum throughput (the case we are interested in), processes will not be able to broadcast new messages immediately. This will result in the creation of a queue of messages at the sender which leads to sending a burst. Thus our model can accurately represent the general case in high load scenarios.

2.3 Related Work

The five following classes of UTO-broadcast protocols have been distinguished in the literature [43]: fixed-sequencer, moving sequencer, privilege-based, communication history, and destination agreement. In this section, we only survey time-free protocols, i.e. protocols that do not rely on physical time, since these are the ones comparable to the LCR protocol. The reason is that the assumption of synchronized clocks is not very realistic in practice, especially since clock skew is hard to detect. We also do not discuss protocols with disk writes as in [92] for instance. Our goal (and that of most of the related work surveyed here) is to optimize the broadcasting of messages at the network level.

2.3.1 Fixed Sequencer

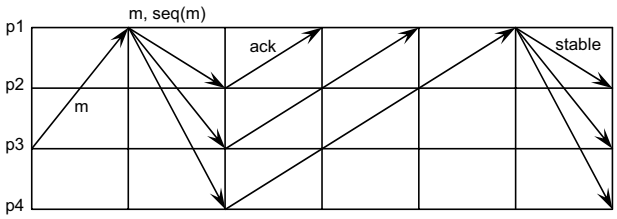


Figure 2.2: Fixed sequencer-based UTO-broadcast.

In a fixed sequencer protocol [64, 8, 26, 54, 21, 97], a single process is elected as the sequencer and is responsible for the ordering of messages (Figure 2.2). The sequencer is unique, and another process is elected as a new sequencer only in the case of sequencer failure. Three variants of the fixed sequencer protocol exist [11], each using a different communication pattern. Fixed sequencer protocols exhibit linear latency with respect to n [42], but poor throughput. The sequencer becomes a bottleneck because it must receive the acknowledgments (acks) from all processes[1] and must also receive all messages to be broadcast. Note that this class of protocols is popular for *non*-uniform total order broadcast protocols since these do not require all processes to send acks back to the sequencer, thus providing much better latency and throughput.

2.3.2 Moving Sequencer

Moving sequencer protocols [31, 96, 66, 40] (Figure 2.3) are based on the same principle as fixed sequencer protocols, but allow the role of the sequencer to be passed from one process to another (even in failure-free situations). This is achieved by a token which carries

[1] Acknowledgments in the fixed sequencer can only be piggy-backed when all processes broadcast messages all the time [42].

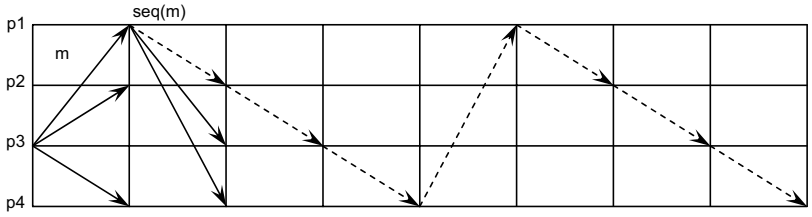


Figure 2.3: Moving sequencer-based UTO-broadcast.

a sequence number and constantly circulates among the processes. The motivation is to distribute the load among sequencers, thus avoiding the bottleneck caused by a single sequencer. When a process p wants to broadcast a message m, it sends it to all other processes. Upon receiving m, processes store it into a *receive* queue. When the current token holder q has a message in its *receive* queue, q assigns a sequence number to the first message in the queue and broadcasts that message together with the token. For a message m to be delivered, it has to be acknowledged by all processes. Acks are gathered by the token. Moving sequencer protocols have a latency that is worse than that of fixed sequencer protocols [43]. On the other hand, these protocols achieve better throughput, although not optimal. Figure 2.3 depicts a 1-to-n broadcast of one message. It is clear from the figure that it is impossible for the moving sequencer protocol to deliver one message per round. The reason is that the token must be received at the same time as the broadcast messages. Thus, the protocol cannot achieve optimal throughput. Even in the n-to-n case, optimal throughput cannot be achieved because at any given time there is only one process which can send messages. Thus, the throughput when all processes broadcast cannot be higher than when only one process broadcasts (in Section 2.5.1 we will show that optimal throughput can only be achieved when all processes broadcast). Note that fixed sequencer protocols are often prefered to moving sequencer protocols because they are much simpler to implement [43].

2.3.3 Privilege-based Protocols

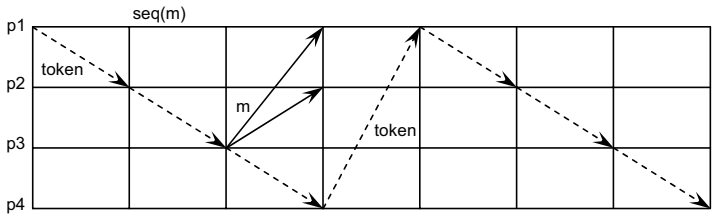


Figure 2.4: Privilege-based UTO-broadcast.

These protocols [51, 39, 47, 4, 56] rely on the idea that senders can broadcast messages only when they are granted the privilege to do so (Figure 2.4). The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process in the form of a token. As with moving sequencer protocols, the throughput when all processes broadcast cannot be higher than when only one process broadcasts.

2.3.4 Communication History-based Protocols

As in privilege-based protocols, communication history-based protocols [84, 77, 48, 82, 81] use sender-based ordering of messages. Nevertheless, they differ by the fact that pro-

cesses can send messages at any time. Messages carry logical clocks that allow processes to observe the messages received by other processes in order to learn when delivering a message does not violate the total order. Communication history-based protocols have poor throughput because they rely on a quadratic number of messages exchanged for each message that is broadcast.

2.3.5 Destination Agreement

In destination agreement protocols, the delivery order results from an agreement between destination processes. Many such protocols have been proposed [29, 20, 75, 53, 7]. They mainly differ by the subject of the agreement: message sequence number, message set, or acceptance of a proposed message order. These protocols have relatively bad performance because of the high number of messages that are generated for each broadcast.

Note that hybrid protocols, combining two different ordering mechanisms have also been proposed [48, 85, 94]. Most of these protocols are optimized for large scale networks instead of clusters, making use of multiple groups or optimistic strategies.

2.4 The LCR Protocol

LCR combines (a) a ring topology for high-throughput dissemination with (b) logical (vector) clocks for message ordering. It is a uniform total order broadcast (UTO-broadcast) protocol exporting two primitives, utoBroadcast and utoDeliver, and ensuring the following four properties:

- **Validity**: if a correct process p_i utoBroadcasts a message m, then p_i eventually utoDelivers m.
- **Integrity**: for any message m, any correct process p_j utoDelivers m at most once, and only if m was previously utoBroadcast by some correct process p_i.
- **Uniform Agreement**: if any process p_i utoDelivers any message m, then every correct process p_j eventually utoDelivers m.
- **Total Order**: for any two messages m and m', if any process p_i utoDelivers m without having delivered m', then no process p_j utoDelivers m' before m.

We detail our LCR protocol in the following. We assume a set $\Pi = \{p_0, \cdots, p_{n-1}\}$ of n processes. Processes are organized in a ring: every process has a predecessor and a successor in the ring: p_0 is before p_1, which is before p_2, etc. We call p_0 "the *first* process in the ring", and p_{n-1} "the *last* process in the ring". We first describe how we totally order messages in LCR. We then describe the behavior of the protocol in the absence of failures and then we describe what happens when there is a group membership change, e.g. a node failing or joining/leaving the system.

2.4.1 Total Order Definition

As we explain later in this section, to broadcast a message, a process sends it to its successor, which itself send its its successor, and so on until every process received the message. We define the total order on messages in LCR as the order according to which messages are received by the last process in the ring, i.e. process p_{n-1}. To illustrate this, consider any two messages m_i and m_j broadcast by processes p_i and p_j, respectively. Assume $i < j$, i.e. p_i is "before" p_j in the ring. In order to determine whether m_i is ordered before m_j, it is enough to know whether p_j had received m_i before sending m_j. This is achieved using

vector clocks: process p_j is equipped with a logical vector clock $\mathcal{C}_{p_j} = (c_k)_{k=[0..n-1]}$. At any time, the value $\mathcal{C}_{p_j}[i]$ represents the number of broadcasts initiated by process p_i that p_j received so far. Before initiating the broadcast of message m_i, process p_i increments the number of broadcasts it initiated (stored in $\mathcal{C}_{p_i}[i]$) and timestamps m_i with its clock. Upon reception of message m_i, process p_j updates its logical clock to take into account message m_i: it increments the value stored in $\mathcal{C}_{p_j}[i]$. If later p_j sends message m_j, it will timestamp m_j with its logical clock that reflects the fact that it received m_i before sending m_j. Every process can thus order m_i and m_j by comparing their clocks. More precisely, let us note $\mathcal{C}_{m_i}$ (resp. $\mathcal{C}_{m_j}$) the logical clock carried by m_i (resp. m_j). Message m_i is ordered before m_j, noted $m_i \prec m_j$, if and only if $\mathcal{C}_{m_i}[i] \leq \mathcal{C}_{m_j}[i]$ when $i < j$, and $\mathcal{C}_{m_i}[i] < \mathcal{C}_{m_j}[i]$ when if $i = j$.

2.4.2 Failure-free Behavior

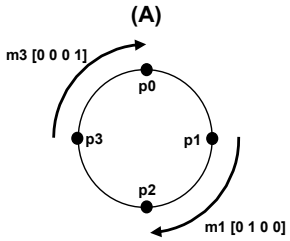
The pseudo-code of the LCR sub-protocol executed in the absence of failures is depicted in Figure 2.5. To broadcast a message m_i, process p_i sends m_i to its successor in the ring. The message is then successively forwarded until it reaches the predecessor of p_i. Processes forward messages in the order in which they receive them. To ensure *total order delivery*, each process ensures, before delivering message m_i, that it will not subsequently receive a message that is ordered before m_i (according to the order defined in the previous section). Moreover, for the sake of *uniform delivery*, each process ensures, before delivering m_i, that all other processes already received m_i (m_i is stable). These guarantees rely on a local list, denoted pending, used by every process to store messages before they are delivered. Messages in pending are totally ordered according to the order defined above. We now explain when messages stored in pending are delivered.

When the predecessor of p_i receives message m_i, it sends an ACK message along the ring. The ACK message is then successively forwarded until it reaches the predecessor of the process that sent it. Upon receiving this ACK message, each process knows (1) that m_i is stable, and (2) that it already received all messages that are ordered before m_i. The first point is obvious. The second point can be intuitively explained as follows. Consider a message m_j that is ordered before m_i. We know, by definition of the total order on messages, that process p_{n-1} will receive m_j before m_i. Provided that messages are forwarded around the ring in the order in which they are received, we know that the predecessor of p_i will receive m_j before sending the ACK for m_i. Consequently, no process can receive the ACK for m_i before m_j. Once a message has been set to stable, it can be delivered as soon as it becomes first in the pending list.

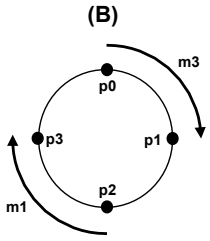
To illustrate the behavior of LCR, consider the following simple example (Figure 2.6) assuming a system of 4 processes. For simplicity of presentation, we consider that the computation proceeds in rounds. The arrays in Figure 2.6 depict the state of the pending list stored at each process at the end of each round. At the beginning of round (A), processes p_1 and p_3 broadcast m_1 and m_3, respectively. Message m_1 is the first message broadcast by p_1 and the latter did not receive any message before broadcasting m_1. Therefore, $\mathcal{C}_{m_1} = [0, 1, 0, 0]$. Similarly, $\mathcal{C}_{m_3} = [0, 0, 0, 1]$. At the end of the round, p_1 and p_2 (resp. p_3 and p_0) have m_1 (resp. m_3) in their pending list. During round (B), p_0 (resp. p_2) forwards m_3 (resp. m_1). At the end of the round, the pending lists of p_1 and p_3 contain two messages: m_1 and m_3. Note that in both lists, m_3 is ordered before m_1. Indeed, processes know that m_3 is ordered before m_1 ($m_3 \prec m_1$) because $p_1 < p_3$ and $\mathcal{C}_{m_1}[1] > \mathcal{C}_{m_3}[1]$, which indicates that when p_3 sent m_3, it had not yet received m_1. During round (C), p_1 (resp. p_3) forwards m_3 (resp. m_1). At the end of the round, all processes have both m_1 and m_3 in their pending list. Moreover, p_0 (resp. p_2) knows that message m_1 (resp. m_3) completed a full round around the ring. Indeed, p_0 (resp. p_2) is the predecessor of the process that sent m_1 (resp. m_3). Consequently, p_0 (resp. p_2) sets m_1 (resp. m_3) to stable. At that time, p_0 (resp. p_2) knows that it will

```
Procedures executed by any process p_i
procedure initialize(initial_view)
    pending_i ← ∅                                          {pending list}
    C[1...n] ← {0,...,0}                                   {local vector clock}
    view ← initial_view

procedure utoBroadcast(m)
    C[i] ← C[i] + 1
    pending ← pending ∪ [m, p_i, C, ⊥]
    Rsend ⟨m, p_i, C⟩ to successor(p_i, view)              {broadcast a message}

upon Rreceive ⟨m, p_j, C_m⟩ do
    if C_m[j] > C[j] then
        if p_i ≠ predecessor(p_j, view) then
            Rsend ⟨m, p_j, C_m⟩ to successor(p_i, view)    {forward the message}
            pending ← pending ∪ [m, p_j, C_m, ⊥]
        else
            pending ← pending ∪ [m, p_j, C_m, stable]      {m is stable}
            Rsend ⟨ACK, p_j, C_m⟩ to successor(p_i, view)  {send an ACK}
            tryDeliver()
        C[j] ← C[j] + 1                                    {update local vector clock}

upon Rreceive ⟨ACK, p_j, C_m⟩ do
    if p_i ≠ predecessor(predecessor(p_j), view) then
        pending[C_m] ← [*, *, *, stable]                   {m is stable}
        Rsend ⟨ACK, p_j, C_m⟩ to successor(p_i, view)      {forward the ACK}
        tryDeliver()

procedure tryDeliver()
    while pending.first = [m, p_k, C_m, stable] do
        utoDeliver(m)                                      {deliver a message}
        pending ← pending − [m, p_k, C_m, stable]
```

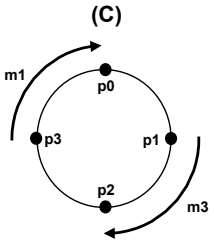
Figure 2.5: Pseudo-code of the LCR protocol.



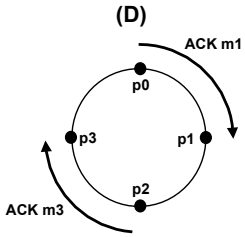
p0 [0 0 0 1]	(m3, 3, [0 0 0 1], .)
p1 [0 1 0 1]	(m1, 1, [0 1 0 0], .)
p2 [0 1 0 0]	(m1, 1, [0 1 0 0], .)
p3 [0 0 0 1]	(m3, 3, [0 0 0 1], .)



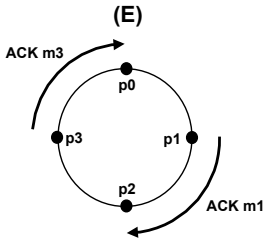
p0 [0 0 0 1]	(m3, 3, [0 0 0 1], .)
p1 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)
p2 [0 1 0 0]	(m1, 1, [0 1 0 0], .)
p3 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)



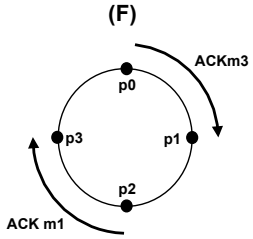
p0 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p1 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)
p2 [0 1 0 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)
p3 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)



p0 [0 1 1 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p1 [0 2 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p2 [0 2 1 1]	(m1, 1, [0 1 0 0], .)
p3 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)



p0 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ ~~(m1, 1, [0 1 0 0], stable)~~
p1 [0 2 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p2 [0 2 1 1]	~~(m1, 1, [0 1 0 0], stable)~~
p3 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)



p0 [0 1 1 1]	
p1 [0 2 0 1]	~~(m3, 3, [0 0 0 1], stable)~~ ~~(m1, 1, [0 1 0 0], stable)~~
p2 [0 2 1 1]	
p3 [0 1 1 1]	~~(m1, 1, [0 1 0 0], stable)~~

Figure 2.6: Illustration of a run of the LCR protocol with 4 processes.

no longer receive any message that is ordered before m_1 (resp. m_3). Indeed, if a message m is ordered before m_1 (resp. m_3), it means, by definition of the total order on messages, that p_3 will receive it before m_1 (resp. m_3). Consequently, p_0 (resp. p_2) will also receive m before m_1 (resp. m_3). Although process p_0 knows that m_1 is stable, it cannot deliver it. Indeed, it first needs to deliver m_3 (which is first in its pending list), which it cannot do. The reason is that it does not know yet whether m_3 is stable or not. Delivering m_3 could violate uniformity. In contrast, process p_2 can deliver m_3 because it is stable and first in its pending list. Process p_2 thus knows that the delivery of m_3 respects total order and uniformity. At the start of round (D), p_0 (resp. p_2) sends an ACK for m_1 (resp. m_3). These ACK messages are forwarded in rounds E and F until they reach the predecessor of the process which initiated the ACK message: for instance, the ACK for message m_1 was initiated by process p_0 in round D. It is forwarded until it reaches process p_3 in round F. Upon reception of these ACK messages (rounds D, E, F), processes set m_1 and m_3 to stable and deliver them as soon as they are first in their pending list.

2.4.3 Group Membership Changes

The LCR protocol is built on top of a group communication system [19]: processes are organized into groups, which they can leave or join, triggering a view change protocol. Faulty processes are excluded from the group after crashing. Upon a membership change, processes agree on a new view. When a process joins or leaves the group, a $view_change$ event is generated by the group communication layer and the current view v_r is replaced by a new view v_{r+1}. This can happen when a process crashes or when a process explicitly wants to leave or join the group. As soon as a new view is installed, it becomes the basis for the new ring topology.

The $view_change$ procedure is detailed in Figure 2.7. Note that when a view change occurs, every process first completes the execution (if any) of all other procedures described in Figure 2.5. It then freezes those procedures and executes the view change procedure. The latter works as follows: every process sends its pending list to all other processes. Upon receiving this list, every process adds to its pending list the messages it did not yet receive. Then the processes send back an ACK_RECOVER message. Processes wait until they receive ACK_RECOVER messages from all processes before sending an END_RECOVERY message to all. When a process receives END_RECOVERY messages from all processes, it can deliver all the messages in its pending list. Thus, at the end of the view change procedure, all pending lists have been emptied, which guarantees that all messages from the old view have been handled.

2.4.4 Correctness

In this section, we prove that LCR is a uniform total order broadcast protocol. We proceed by successively proving that LCR ensures the four properties mentioned at the beginning of Section 2.4: validity, integrity, uniform agreement and total order.

Lemma 2.4.1 (Validity) *If any correct process p_i* utoBroadcasts *a message m, then it eventually* utoDelivers *m.*

Let p_i be a correct process and let m_i be a message broadcast by p_i. This message is added to p_i's pending list (Line 7 of Figure 2.5). If there is a membership change, p_i being a correct process, it will be in the new view. Consequently, the view change procedure guarantees that p_i will deliver all messages stored in its pending list (Line 6 of Figure 2.7). It will thus deliver m_i. Let us now consider the case when there is no membership change. All processes (including p_i) will eventually set m_i to stable (Line 21 of Figure 2.5). This is due to the fact that m_i will be forwarded along the ring (because $\mathcal{C}_{m_i}[i]$ is higher than the ith value

Procedures executed by any process p_i

1: **upon** view_change(new_view) **do**
2: Rsend $\langle$RECOVER, $p_i, pending\rangle$ to all $p_j \in new_view$
3: Wait until received $\langle$ACK_RECOVER$\rangle$ from all $p_j \in new_view$
4: Rsend $\langle$END_RECOVERY$\rangle$ to all $p_j \in new_view$
5: Wait until received $\langle$END_RECOVERY$\rangle$ from all $p_j \in new_view$
6: forceDeliver()
7: $view \leftarrow new_view$

8: **upon** Rreceive $\langle$RECOVER, $p_j, pending_{p_j}\rangle$ **do**
9: **for each** $[m, p_l, \mathcal{C}_m, *] \in pending_{p_j}$ **do**
10: **if** $\mathcal{C}_m[l] > \mathcal{C}[l]$ **then**
11: $pending \leftarrow pending \cup [m, p_l, \mathcal{C}_m, \perp]$
12: Rsend $\langle$ACK_RECOVER$\rangle$ to p_j

13: **procedure** forceDeliver()
14: **for each** $[m, p_k, \mathcal{C}_m, *] \in pending$ **do**
15: utoDeliver(m) *{deliver a message}*
16: $pending \leftarrow pending - [m, p_k, \mathcal{C}_m, *]$
17: $\mathcal{C}[k] \leftarrow \mathcal{C}[k] + 1$ *{update local vector clock}*

Figure 2.7: Pseudo-code of the view_change procedure.

stored in the clock of each process) until it reaches the predecessor of p_i. The latter marks m_i as stable (line 15 of Figure 2.5) and sends an ACK to its successor in the ring containing $\mathcal{C}_{m_i}$. Similarly to m_i, the ACK message is forwarded along the ring (line 22 of Figure 2.5) until it reaches the predecessor of the predecessor of p_i. Upon receiving the ACK message, each process marks m_i as stable. When p_i sets m_i to stable, its pending list starts with a (possibly empty) set of messages m such that $m \prec m_i$ and m has not been yet delivered by p_i. Let us call $undelivered$ this set of messages. Let us first remark that this set cannot grow. Consider, for instance, the case of a message m_j sent by a process p_j that precedes m_i (i.e. $m_j \prec m_i$). We know that p_j sent m_j before receiving m_i. Consequently, the predecessor of p_i will receive m_j before receiving m_i, and thus before sending the ACK for m_i. As each process forwards messages in the order in which it receives them, we know that p_i will necessarily receive m_j before receiving the ACK for message m_i. Let us now consider every message in $undelivered$. As there is no membership change, the same reasoning as the one we did for m_i applies to messages in $undelivered$: every process will eventually set these messages to stable. Consequently, all messages in $undelivered$ will be delivered. Message m_i will thus be first in $pending$ and marked as stable. It will thus also be delivered by p_i.

Lemma 2.4.2 (Integrity) *For any message m, any process p_j* utoDelivers *m at most once, and only if m was previously* utoBroadcast *by some process p_i.*

No spurious message is ever utoDelivered by a process as we assume only crash failures. Thus, only messages that have been utoBroadcast are utoDelivered. Moreover, each process keeps a vector clock $\mathcal{C}$, which is updated in such a way that we are sure that every message is only delivered once. Indeed, if there is no membership change, Lines 10 and 18 of Figure 2.5 guarantee that no message can be processed twice by p_j. Similarly, when there is a membership change, Line 10 of Figure 2.7 guarantees that process p_j will not deliver messages twice. Moreover, Line 17 of Figure 2.7 guarantees that p_j's vector clock is updated after the membership change, thus preventing the future delivery of messages that have been delivered during the $view_change$ procedure.

Lemma 2.4.3 (Uniform Agreement) *If any process p_i* utoDelivers *any message m in the current view, then every correct process p_j in the current view eventually* utoDelivers *m.*

Let m_k be a message sent by process p_k and let p_i be a process that delivered m_k in the current view. There are two cases to consider. In the first case, p_i delivered m_k during a membership change. This means that p_i had m_k in its pending list before executing line 15 of Figure 2.7. Since all correct processes exchange their pending list during the view change procedure, we are sure that all correct processes that did not deliver m_k before the membership change will have it in their pending list before executing line 6 of Figure 2.7. Consequently, all correct processes in the current view will deliver m_k. The second case to consider is when p_i delivered m_k outside of a membership change. The protocol ensures that m_k did a full round around the ring before being delivered by p_i: indeed p_i can only deliver m_k after having set it to stable, which either happens when it is the predecessor of p_k in the ring or when it receives an ACK for message m_k. Consequently, all processes stored m_k in their pending list before p_i delivered it. If a membership change occurs after p_i delivered m_k and before all other correct processes delivered it, the protocol ensures that all correct processes that did not yet deliver m_k will do it (Line 6 of Figure 2.7). If there is no membership change after p_i delivered m_k and before all other processes delivered it, the protocol ensures that an ACK for m_k will be forwarded around the ring, which will cause all processes to set m_k to stable. Each correct process will thus be able to deliver m_k as soon as m_k will be first in the pending list (Line 26 of Figure 2.5). The protocol ensures that m_k will become first eventually. The reasons are the following: (1) the number of messages that are before m_k in the pending list of every process p_j is strictly decreasing, and (2) all messages that are before m_k in the pending list of a correct process p_j will become stable eventually. The first reason is a consequence of the fact that once a process p_j sets message m_k to stable, it can no longer receive any message m such that $m \prec m_k$. Indeed, a process p_l can only produce a message $m_l \prec m_k$ before receiving m_k. As each process forwards messages in the order in which it received them, we are sure that the process that will produce an ACK for m_k will have first received m_l. Consequently, every process setting m_k to stable will have first received m_l. The second reason is a consequence of the fact that for every message that is utoBroacast in the system, the protocol ensures that an ACK will be forwarded around the ring (Lines 16 and 22 of Figure 2.5), implying that all correct processes will mark the message as stable. Consequently, all correct processes will eventually deliver m_k.

Lemma 2.4.4 (Total Order) *For any two messages m and m', if any process p_i* utoDelivers *m without having delivered m', then no process p_j* utoDelivers *m' before m.*

We prove the lemma by contradiction. Let m and m' be any two messages and let p_i be a process that utoDelivers m without having delivered m'. Consider a process p_j that utoDelivers m' before delivering m. Let us denote t_i the time at which p_i delivered m and t_j the time at which p_j delivered m'. Let us first note that the protocol ensures that at time t_i (resp. t_j), all processes have already received m (resp. m'). Indeed, when there is no membership change, a message can only be delivered after it is set to stable, which requires the message to have done a full round around the ring. When there is a membership change, the protocol ensures (by broadcasting messages stored in pending lists and waiting for all processes to have received all broadcast before delivering any message) that all processes have a consistent pending list before delivering messages (Lines 2 to 5 of Figure 2.7).

Case 1: $t_j \leq t_i$. It follows that at time t_i, process p_i had already received m'. Consequently, $m \prec m'$, otherwise, p_i would have delivered m' first. We will show that in that case, we are sure that p_j received m before m', thus contradicting the fact that it delivered m' before m. Let us note p_k the process that utoBroadcast m. Provided $m \prec m'$, we know that p_k sent m before receiving m'. There are two cases to consider: if there is no membership change before p_j delivers m', as each process forwards messages in the order in which it

received them, we are sure that p_j will receive m before it can set m' to stable. The second case is when there is a membership change before p_j delivers m'. Process p_j could not receive m before the membership change, otherwise, it would not deliver m' before m (since we know that $m \prec m'$). Provided that p_i delivers message m, we know that this can only happen during the membership change. Indeed, the message can not have been delivered before, otherwise p_j would have received it. Consequently, at the beginning of the view change procedure, p_i has m in its pending list and will send it to all processes. Consequently, p_j will receive m during the view change procedure. In both cases, we are sure that p_j received m before delivering m', which is in contradiction with the fact that it delivered m' before m provided that $m \prec m'$.

Case 2: $t_i < t_j$. It follows that at time t_j, process p_j had already received m. Consequently, $m' \prec m$, otherwise, p_j would have delivered m first. With a similar reasoning as the one we did for case 1, we know that p_i received m' before delivering m, which is in contradiction with the fact that it delivered m without delivering m' provided that $m' \prec m$.

Theorem 2.4.5 *LCR is a uniform total order broadcast protocol.*

By lemmas 2.4.1, 2.4.2, 2.4.3, and 2.4.4, we can derive the very fact that the LCR protocol ensures validity, integrity, uniform agreement, and total order. Thus, it is a uniform total order broadcast protocol.

2.5 Theoretical Analysis

This section analyzes several key aspects of LCR's performance from a theoretical perspective. The performance of LCR is evaluated in failure free runs which we expect to be the common case. We prove that LCR is throughput optimal in such case. Then we discuss its fairness.

2.5.1 Throughput

In this section we show that the throughput of LCR is optimal and that no other broadcast protocol (even with weaker consistency guarantees) can obtain strictly higher throughput. We do this by proving an upper bound on the performance of any broadcast protocol and show that LCR matches this bound.

Theorem 2.5.1 (Maximum throughput for any broadcast protocol) *For a broadcast protocol in a system with n processes in the round-based model introduced in Section 2.2.2, the maximum throughput μ_{max} in completed broadcasts per round is:*

$$\mu_{max} = \begin{cases} n/(n-1) & \textit{if there are n senders} \\ 1 & \textit{otherwise} \end{cases}$$

We first consider the case with n senders. Each broadcast message must be received at least $n-1$ times in order to be delivered. The model states that at each round at most n messages can be received. Thus, for n processes to broadcast a message, a minimum of $n-1$ rounds are necessary. Therefore, on average, at most $n/(n-1)$ broadcasts can be completed each round. In the case with less than n senders it is sufficient to look at a non sending process. Such a process can receive at most 1 message per round and since it doesn't broadcast any messages itself, it can deliver at most 1 message per round. Since the throughput is defined as the number of *completed* broadcasts per round, the maximum throughput with less than n senders is equal to 1.

Determining the throughput of LCR is straightforward: processes receive one message per round and the acknowledgements are piggy-backed. Thus LCR allows each process to deliver one message per round if there is at least one sender. When there are n senders, each process can deliver one message per round broadcast by other processes in addition to its own messages. LCR thus matches the bound of Theorem 2.5.1 and is theoretically throughput optimal. Thus, from a throughput perspective, the strong uniform total order guarantees provided by LCR are free.

2.5.2 Fairness

Even though the throughput of LCR as described thus far is optimal, there is still a problem. Consider two processes p_1 and p_2 that are neighbors on the ring. If p_1 is continuously broadcasting messages and p_2 systematically forwards p_1's messages, then p_2 cannot broadcast its own messages. Consequently, the protocol is not *fair*.

Fairness captures the fact that each process has an equal opportunity of having its messages delivered by all processes. Intuitively, the notion of fairness means that in the long run no single process has priority over other processes when broadcasting messages. Said differently, when two processes want to broadcast a large number of messages during a time interval τ, then each process should have approximately the same number of messages delivered by all processes during τ.

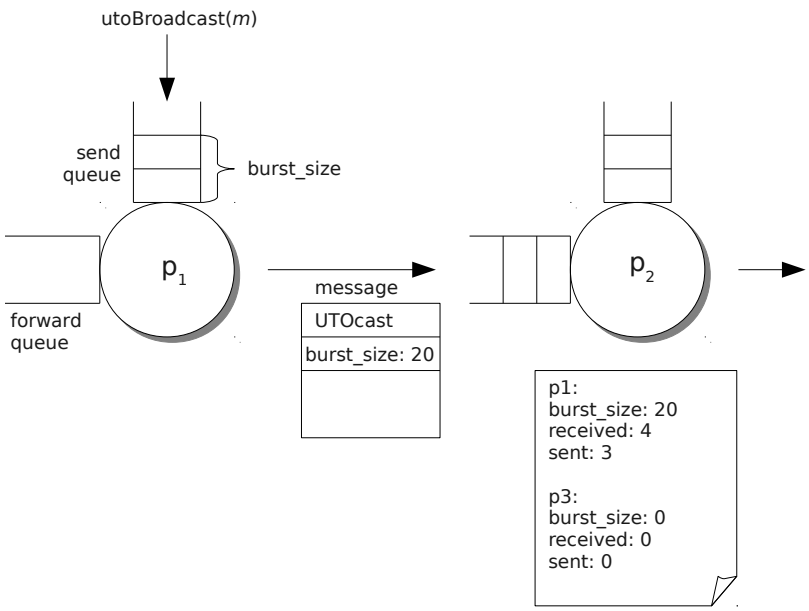


Figure 2.8: Illustration of the fairness mechanism as implemented in LCR. Each process has two queues (send and forward) and uses the burst_size, received, and sent variables to determine whether to forward messages or send its own.

The mechanism for ensuring fairness in LCR acts locally at each process. If a process wishes to broadcast a new message, it must decide whether to forward a message received from its predecessor or to send its own. Figure 2.8 provides an illustration of the fairness mechanism as implemented in LCR. Processes have two queues: send and forward. Processes put broadcast requests coming from the application level in their send queue. Messages received from predecessors that need to be forwarded are buffered in the forward queue. When a process has a burst of messages to send (i.e. it has more than one message in its send

queue), it piggybacks on the first message it sends an integer burst_size representing the number of messages currently stored in its send queue. Each process keeps a data structure which stores 3 integers per process in the ring: burst_size, received and sent. For each process p_i, the burst_size variable is updated every time p_i piggybacks a new burst_size value on a message it sends. The received variable keeps track of the number of messages that the process received from p_i since p_i's burst_size has been updated. The sent variable keeps track of the number of messages that the process sent since p_i's burst_size has been updated. When the received variable is equal to the burst_size, the three integers kept for process p_i are reset (i.e. the burst initiated by p_i is finished). When the process wants to send a message, it retrieves the first message in the forward list. Assume that this message has been sent by process p_j. The process only sends its own message if the received integer stored for p_j is higher or equal than the sent integer stored for p_j, which intuitively means that since p_j started its last burst, the process initiated less broadcasts than p_j.

2.5.3 Latency

The theoretical latency of broadcasting a single message is defined as the number of rounds that are necessary from the initial broadcast of message m until the last process delivers m. The latency of LCR is equal to $2n - 2$ rounds.

2.6 Experimental Evaluation

This section compares the performance of LCR to that of two existing group communication systems: JGroups and Spread. Spread ensures uniform total order delivery of messages, whereas JGroups only guarantees *non-uniform* total order delivery. The experiments only evaluate the failure free case because failures are expected to be sufficiently rare in the targeted environment. Furthermore, the view change procedure that is used in LCR is similar to that of other total order broadcast protocols [43].

We first present the experimental setting we used in the experiments and then study various performance metrics: throughput, response time, fairness, and CPU consumption. In particular, we show that LCR always achieves a significantly better throughput than Spread and JGroups when all processes broadcast message. We also show that when only one process broadcasts messages, LCR outperforms Spread and has similar performance than JGroups. Regarding response time, we show that LCR exhibits a higher response time than Spread and JGroups when there is only one sender in the system. In contrast, it outperforms both protocols when all processes broadcast messages. Finally, we show that LCR and Spread are both fair and have a low CPU consumption. This contrasts with JGroups that is not fair and has a higher CPU consumption than Spread and LCR.

2.6.1 Experimental Setup

The experiments were run on a cluster of machines with a $1.66GHz$ bi-processor and $2GB$ RAM. Machines run the Linux $2.6.30 - 1$ SMP kernel and are connected using a Fast Ethernet switch. The raw bandwidth over IP is measured with Netperf [63] between two machines and displayed in Table 2.1.

The LCR protocol is implemented in C ($\approx$ 1000 lines of code). It relies on the Spread toolkit to provide a group membership layer and uses TCP for communication between processes. As explained in [46], it is reasonable to assume, in an low-latency cluster, that when a TCP connection fails, the server on the other side of the connection failed. It is thus easy to implement the abstraction of a perfect failure detector [28]. Moreover, using TCP, it is not

Protocol	Bandwith
TCP	$93Mb/s$
UDP	$93Mb/s$

Table 2.1: Raw network performance measured using Netperf.

necessary to implement a message retransmission mechanism: a message sent from a correct process to another correct process will be delivered eventually.

The implementation of LCR is benchmarked against two industry standard group communication systems:

- *Spread*. We use Spread version 4.1 [3]. The message type was set to `SAFE_MESS` which guarantees uniform total order. Spread implements a privilege-based ordering scheme (see Section 2.3.3). A Spread daemon was deployed on each machine. All daemons belong to the same Spread segment. Spread was tuned for bursty traffic according to Section 2.4.3 of the Spread user guide [89]. Our benchmark uses the native C API provided by Spread.

- *JGroups*. We use JGroups version 2.7.0 [14] with the Java HotSpot Server 1.6.0_16 virtual machine. We use the "`sequencer`" stack that contains the following protocols: `UDP, PING, MERGE2, FD_SOCK, FD_ALL, VERIFY_SUSPECT, BARRIER, pbcast.NAKACK, UNICAST, pbcast.STABLE, VIEW_SYNC, pbcast.GMS, SEQUENCER, FC, FRAG2, STATE_TRANSFER`. This stack provides *non uniform* total ordering. It implements a fixed-sequencer ordering scheme without acknowledgements (see Section 2.3.1).

All experiments we present in this section start with a warm-up phase, followed by a phase during which performance are measured. Finally, there is a cool-down phase without measurements. The warm-up and cool-down phases last 5 minutes. The measurement phase lasts 10 minutes.

2.6.2 Throughput

To assess the throughput of total order broadcast protocols, we use the following benchmark: k processes out of the n processes in the system broadcast messages at a predefined throughput (we call this experiment k-to-n broadcast). Each message has a fixed size, which is a parameter of the experiment. Each process periodically computes the throughput at which it delivers messages. The throughput is calculated as the ratio of delivered bytes over the time elapsed since the end of the warm-up phase. The plotted throughput is the average of the values computed by each process.

We first want to confirm our claim that LCR achieves optimal throughput. Figure 2.9 shows the results of an experiment with $n = 5$ processes. We vary the number k of broadcasting processes (X axis). The size of messages broadcast by processes is 10kB. Moreover, each broadcasting processes produce messages at the maximum throughput it can sustain. We first execute LCR without enabling the fairness mechanism described in Section 2.5.2. We observe that the throughput obtained by LCR is far from optimal: in practice, a theoretical throughput of 1 should be equal to the raw link speed between the processes, i.e. 93Mbit/s as shown in Table 2.1. The reason why the throughput is not optimal is that when the fairness mechanism is disabled, senders can broadcast more messages than can be delivered, resulting in overflowing network buffers and the data structures maintained by each process.

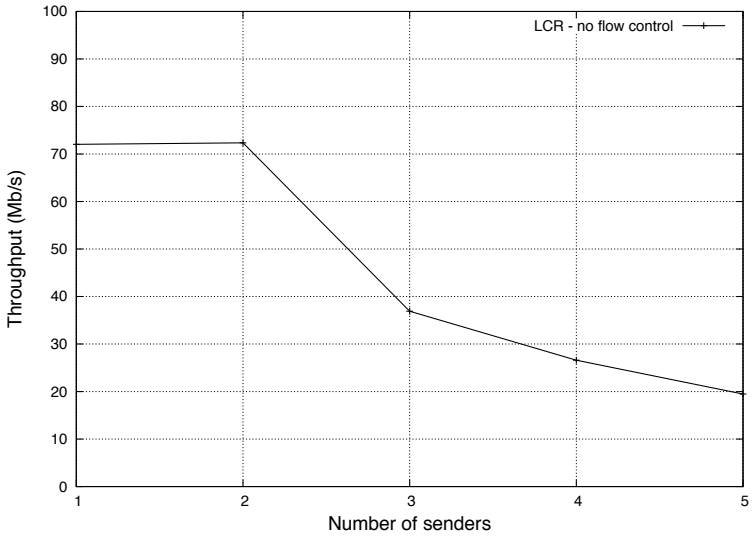


Figure 2.9: LCR throughput with the fairness mechanism disabled in a system with 5 processes. Buffers are quickly saturated, which explains the low throughput.

Figure 2.10 depicts the throughput obtained by LCR when the fairness mechanism is enabled. The throughput clearly improves and is now close to optimal. The reason why the throughput improves is that the fairness mechanism throttles the senders, and does thus prevent them from injecting too many new messages into the ring. Note that the throughput with 5 senders is higher than with fewer senders, the reason being LCR's theoretical throughput of $n/(n-1)$ when all processes are senders. Finally, note that in all subsequent experiments, the fairness mechanism is enabled.

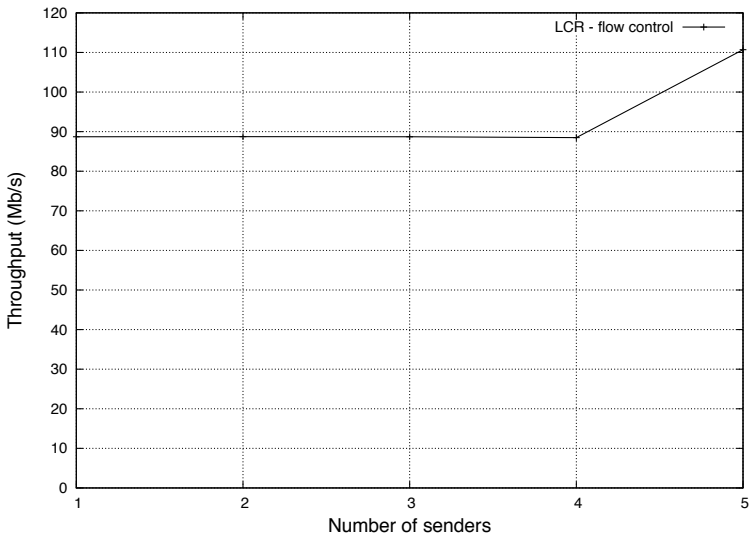


Figure 2.10: LCR throughput with the fairness mechanism enabled in a system with 5 processes. Senders are throttled, which improves the throughput.

The next experiments we present (Figure 2.11 and 2.12) measure the impact of varying the message size on the throughput of LCR, Spread and JGroups for a system with 5 processes.

In the experiment depicted in Figure 2.11, only one process broadcasts messages, whereas all processes broadcast messages in the experiment depicted in Figure 2.12. In both cases, we can observe that if the messages are too small, the throughput of all protocols suffers. This is due to the cost of ordering which remains constant despite a decrease in payload size. We can nevertheless observe that LCR achieves significantly better performance with small messages than Spread and JGroups. To improve performance, it is possible to batch small messages together into bigger messages when the load on the system is high as suggested in [52]. For all further experiments the message size is set to $10kB$, which is the optimal message size for the three protocols in both the 1-to-n and n-to-n cases.

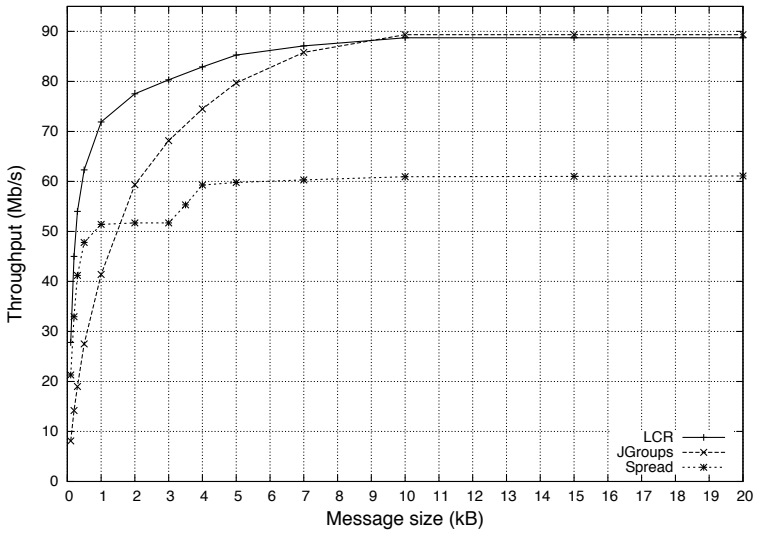


Figure 2.11: Throughput with respect to message size for a system of 5 processes with one sender.

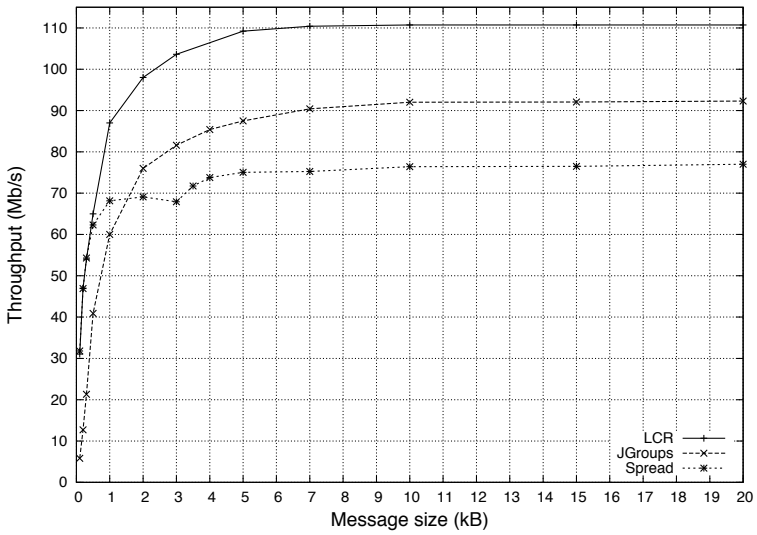


Figure 2.12: Throughput with respect to message size for a system of 5 processes with 5 senders.

Having studied the impact of message size, we now study how the throughput evolves as a

function of the number of processes. Figure 2.13 plots the results of an experiment consisting in 1-to-n broadcasts of $10kB$ messages. The throughput achieved by LCR and JGroups is close to optimal and almost constant despite the increasing number of processes. Note that JGroups does not provide uniformity, and does thus implement a very simple communication pattern: the sender sends its messages to the sequencer, which multicast them to all other processes. Spread's throughput suffers a bit more from increasing the number of processes. This can be explained by the fact that Spread uses a token to order messages and ensure uniformity. Increasing the number of processes increases the time it takes for the token to circulate among all processes.

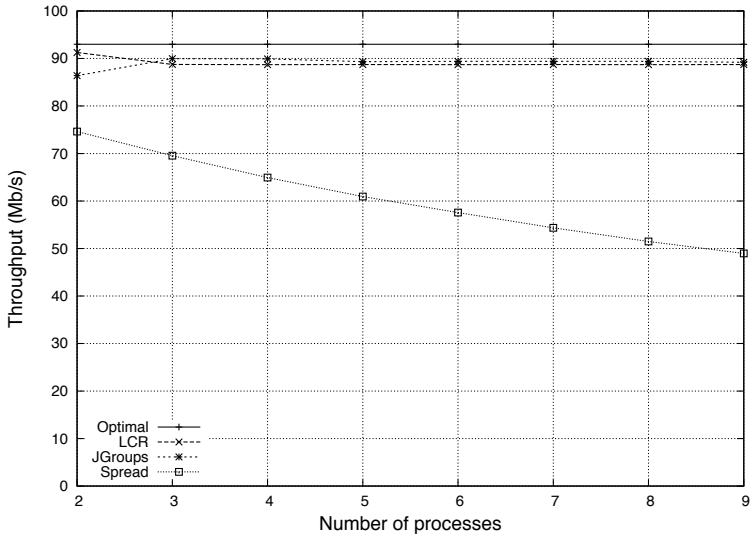


Figure 2.13: 1-to-n throughput comparison. The optimal line is constant at $93Mb/s$.

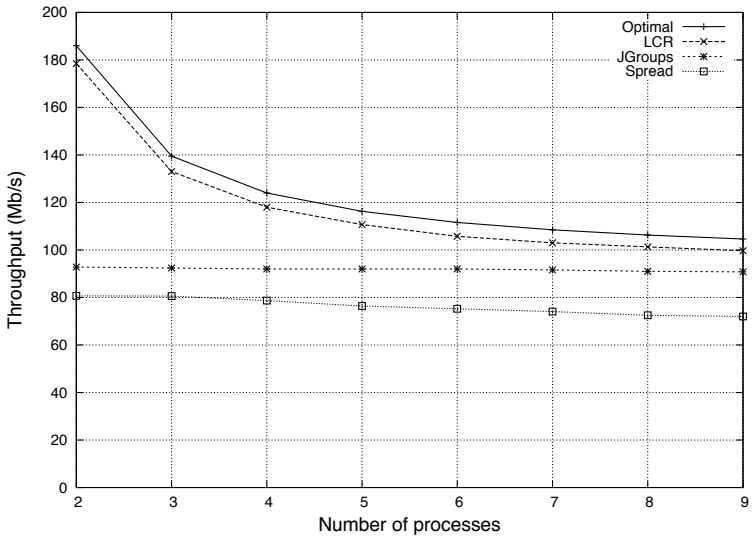


Figure 2.14: n-to-n throughput comparison. The optimal line is calculated as $n/(n-1) * 93Mb/s$.

Figure 2.14 plots the throughput as a function of the number of processes in a system

where all processes broadcast $10kB$ messages. The optimal line for best effort broadcast ($n/(n-1)$ times the maximum link speed of $93Mb/s$) is plotted as a reference. We can first observe that the throughput of LCR is very close to optimal. Since the optimality line is calculated for best effort broadcast and LCR provides uniform total order broadcast, we can conclude that the ordering, reliability and uniformity properties of LCR are effectively almost free. The throughput of JGroups is almost constant (at $92Mb/s$) and only slightly better than the throughput achieved with only one sender (Figure 2.13). This can be easily explained by the fact that the throughput is limited by the throughput at which the sequencer can broadcast messages to other processes in the system (using IP multicast). The throughput achieved by Spread is better than in the 1-to-n case due to the fact that every process makes use of the token to broadcast the messages it produces. Nevertheless, as in the 1-to-n case, the throughput slightly decreases when the number of processes increases due to the increasing cost of ensuring uniformity (cost that JGroups does not have).

To summarize, we can say that LCR is the only protocol that fully exploits available network links when all processes broadcast messages, which we believe is the common case in many applications. It does thus sustain a significantly higher throughput than other protocols. For instance, the gain in throughput in a system with 4 processes is of about 28% compared to JGroups and of about 49% compared to Spread.

2.6.3 Response Time

In this section, we evaluate the response time of LCR, Spread and JGroups in the 1-to-n and n-to-n cases. We setup a system with 5 processes. We vary the throughput at which the sender processes inject new messages. The size of messages that are broadcast is $10kB$. During the measurement phase, for every message m it broadcasts, the sender evaluates the elapsed time between the broadcast and the delivery of m. For each protocol, we stop the curve when the injected load is higher than the throughput the protocol is able to sustain.

Figure 2.15 depicts the results obtained with one sender. In order to evaluate the general case where every process could be the sender, we do not colocate the sender and the sequencer in JGroups. We observe that Spread exhibits a consistently lower response time than JGroups and LCR ($3ms$ against $4ms$ for JGroups and $5ms$ for LCR when the input load is below $10Mb/s$). The fact that LCR exhibits higher response time in the 1-to-n case is not surprising provided that processes are organized in a ring topology. Interestingly, LCR's response time does not degrade when the input load increases. This is in contrast with the response time of JGroups which increases when the input load is greater than $10Mb/s$ (a similar behavior is observed in the n-to-n case). Finally, a last remark we can make is that, although providing uniform delivery of messages, Spread achieves better response time than JGroups. This is due to the fact that Spread uses a token. Once the sender process obtains the token, it can send messages in burst, thus decreasing the average response time. In JGroups, the sender always needs to first send the message to the sequencer, which will then multicast the message to other processes. Finally, note that the fact that the sender in Spread can send multiple messages in burst when it owns the token also explains why the response time slightly decreases when the load gets higher (between $20Mb/s$ and $40Mb/s$).

Figure 2.16 depicts the results obtained with n senders (note that the scale used on the Y axis is different than in Figure 2.15). LCR and JGroups exhibit a lower response time than Spread ($5.2ms$ for JGroups against $5.4ms$ for LCR and $7ms$ for Spread when the input load is below $10Mb/s$). The reason is that with Spread, senders must wait to have the token before sending their messages. As every process has messages to send, it takes a longer time to obtain the token. Note that the response time of Spread is almost constant, and only slightly increases when the maximum delivery throughput is reached. The same remark applies to LCR for which the response time only slightly increases when the input load is higher than $80Mb/s$ but remains low until the maximum delivery throughput is reached (up

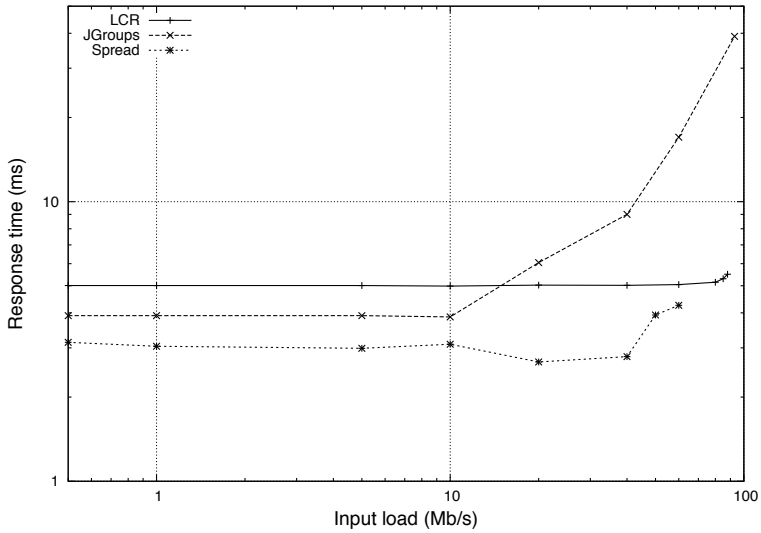


Figure 2.15: 1-to-n response time comparison.

to $108Mb/s$, the response time is below $7.5ms$). In contrast, the response time of JGroups starts degrading when the input load is higher than $10Mb/s$. It becomes high for input loads higher than $70Mb/s$. This is probably due to the fact that the sequencer simultaneously receives messages from all other processes, which fills up its network buffers and increases the time it takes for a message to be sequenced.

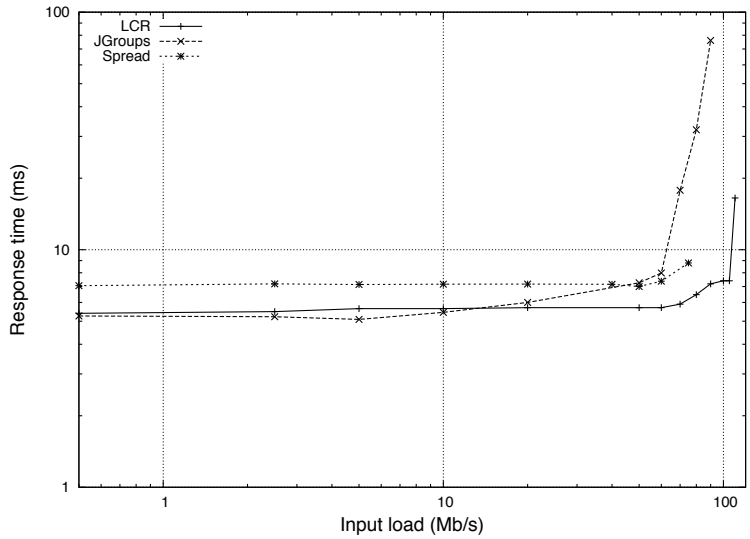


Figure 2.16: n-to-n response time comparison.

To summarize, LCR exhibits a higher response time than other protocols when there is only one sender. When there are multiple senders, LCR's response time equals or outperforms those of other protocols. More precisely, it exhibits better response time than Spread and JGroups (when the load is higher than $20Mb/s$), and similar response time than JGroups (when the load is below $20Mb/s$), but contrarily to the latter, it ensures uniform delivery of messages and the response time does not degrade when the input load increases.

2.6.4 Fairness

We evaluate the fairness of LCR, JGroups and Spread as follows: we setup a system with 5 processes, of which 3 are senders. Each sender continuously broadcasts messages. At the end of the measurement phase, every process computes the percentage of messages it delivered that were issued by each of the 3 senders (called p_1, p_2 and p_3). In the case of JGroups, process p_1 is also the sequencer. The results are depicted in Figure 2.17. We can first observe that both LCR and Spread are fair: each process delivers 33% of messages from each sender. Concerning LCR, this is due to the fact that it implements the fairness mechanism described in Section 2.5.2. Concerning Spread, this is due to the fact that, on average, each sender owns the token for the same amount of time. We can also observe that JGroups is not fair. This comes from the fact that processes p_2 and p_3 first need to send their messages to the sequencer p_1, whereas the latter can directly broadcast the messages it produces. This induces a significant unbalance: 50% of the messages that are delivered have been broadcast by p_1.

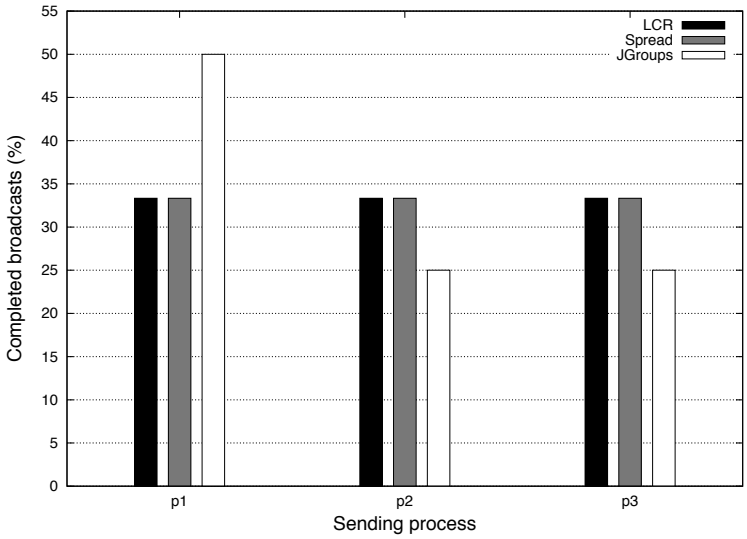


Figure 2.17: Fairness assessment of the LCR, Spread and JGroups protocols. Experiments were performed with 5 processes and 3 senders.

2.6.5 CPU Usage

The last performance metric we evaluate is the CPU usage of LCR, Spread and JGroups under high load. During the experiment, the CPU usage of all active protocol threads was periodically logged, added up and averaged. We study both the 1-to-n and n-to-n cases.

The experiment in Figures 2.18, 2.19, and 2.20 plot the CPU usage measured in a system with 5 processes, of which one broadcasts $10kB$ messages. The X axis represents the message size (in kB). The Y axis represents the CPU consumption (in $\%$). To ease the comparison between the various protocols, the three graphs use the same scale. In the case of JGroups, the sender was not sequencer in order to be able to isolate the CPU consumption of the sequencer, the sender and receiver processes, respectively. The first remark we can make is that among the three protocols, JGroups has the highest CPU consumption. In particular, the sequencer process consumed more than 55% of the CPU in all experiments we performed. The sender performs also significantly more work than receiver processes due to the fact that it needs both to send and receive the messages it broadcasts. Spread and LCR have a CPU

usage that is very reasonable: it is systematically below 30% with messages bigger than $1kB$. In LCR, it is interesting to notice that the sender has less work to do than other processes. The reason is that the sender does not receive messages to forward; it only receives acks. In contrast, the sender and the receivers in Spread use the same percentage of CPU time. Finally, it is interesting to note that Spread uses more CPU than LCR for large messages.

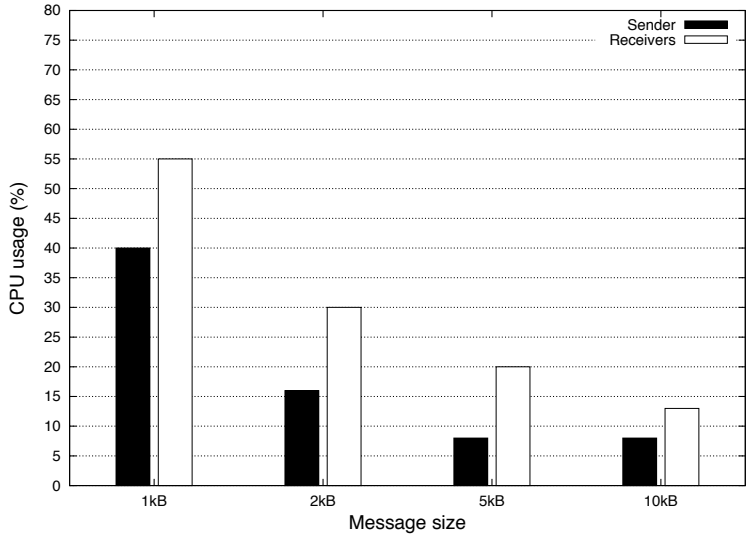


Figure 2.18: CPU usage during high load 1-to-n broadcasts of the LCR protocol.

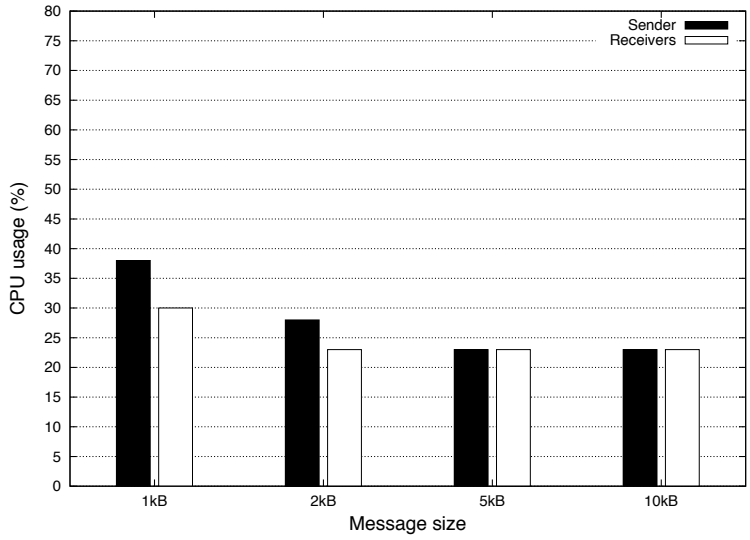


Figure 2.19: CPU usage during high load 1-to-n broadcasts of the Spread protocol.

The experiment in Figure 2.21, 2.22, and 2.23 plot the CPU usage measured in a system with 5 processes, each broadcasting $10kB$ messages. As was the case with only one sender, we observe that JGroups consumes more CPU than other protocols. Interestingly, the consumption of the sequencer is a bit lower than what it was in the previous experiment. We

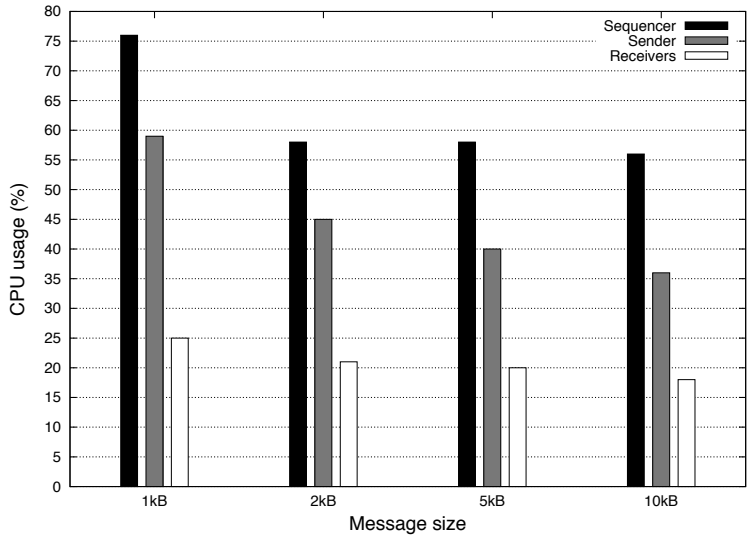


Figure 2.20: CPU usage during high load 1-to-n broadcasts of the JGroups protocol.

explain this by the fact that in this experiment, the sequencer itself broadcasts messages, thus reducing the number of messages it gets from other processes, and thus its CPU consumption. We can also remark that Spread and LCR have a slightly higher CPU usage than in the previous case. This is explained by the fact that both protocols handle more messages (they achieve higher throughput) in the n-to-n case than in the 1-to-n case, thus requiring higher CPU usage. Finally, we observe that, similarly to the previous case, LCR consumes less CPU than Spread when messages are larger.

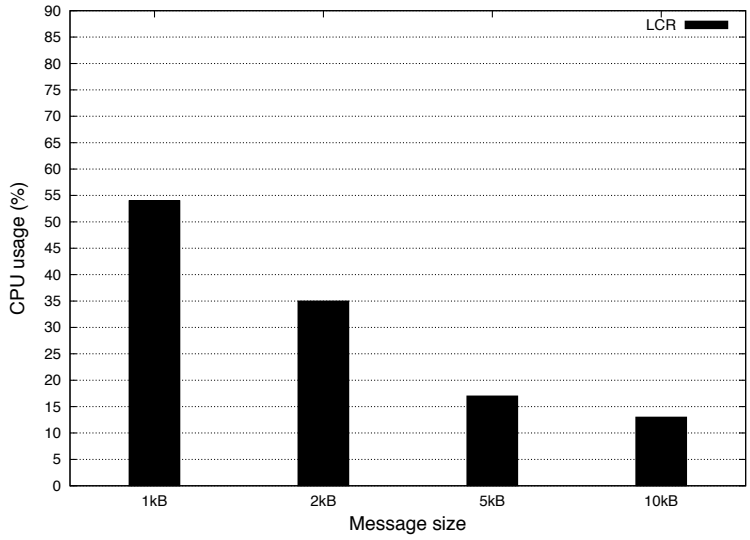


Figure 2.21: CPU usage during high load n-to-n broadcasts of the LCR protocol.

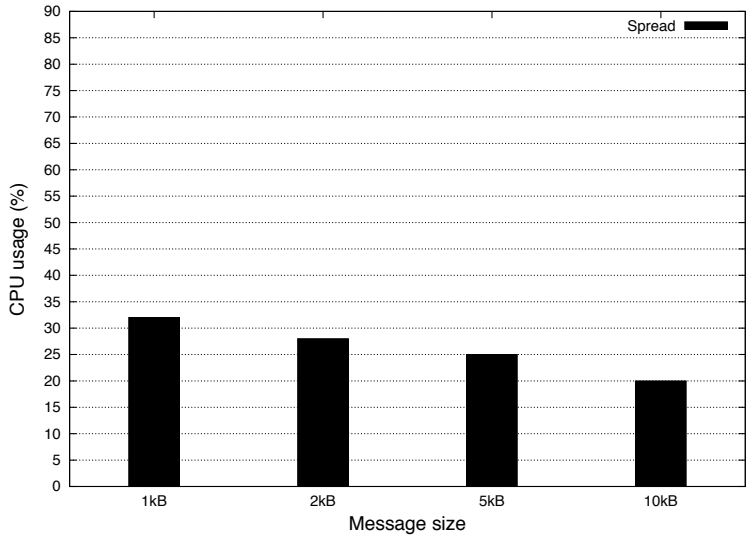


Figure 2.22: CPU usage during high load n-to-n broadcasts of the Spread protocol.

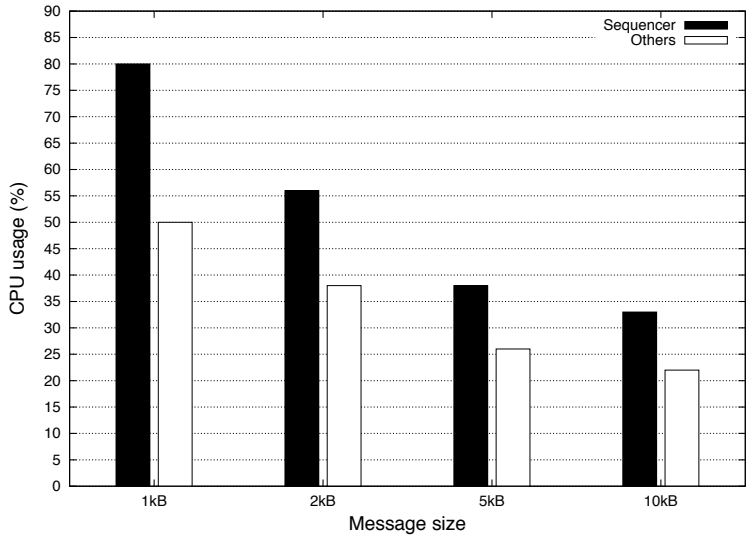


Figure 2.23: CPU usage during high load n-to-n broadcasts of the JGroups protocol.

2.7 Conclusion

In this chapter, we have presented LCR [60, 61, 73], a uniform total order broadcast protocol that can be used as the main communication block of a state machine replication scheme to achieve software-based fault-tolerance.

LCR is the first uniform total order broadcast protocol that is throughput optimal in failure-free periods. In short, throughput optimality captures the ability to deliver the largest possible number of message broadcasts, regardless of message broadcast patterns. This notion is precisely defined in a round-based model of computation which accurately captures message passing interaction patterns over clusters of homogeneous machines interconnected by a fully switched LAN. LCR is based on a ring topology and only relies on point-to-point inter-process communication. LCR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes. Performance benchmarks showed that LCR had a very high throughput in all cases, while exhibiting very reasonable response time. Moreover, benchmarks showed that LCR has a low CPU usage.

Chapter 3

ABsTRACT: a Modular Approach to Building BFT Protocols

This chapter presents Abstract [58, 95], a new abstraction to simplify the design, proof and implementation of BFT protocols. We treat a BFT protocol as a composition of instances of our abstraction. Each instance is developed and analyzed independently. To illustrate our approach, we first show how, with our abstraction, the benefits of a BFT protocol like Zyzzyva could have been developed using less than 24% of the actual code of Zyzzyva. We then present *Aliph*, a new BFT protocol that outperforms previous BFT protocols both in terms of latency (by up to 30%) and throughput (by up to 360%).

3.1 Introduction

In this chapter, we focus on the most robust class of state machine replication (SMR) protocols. These protocols tolerate (a) arbitrarily large periods of asynchrony, during which communication delays and process relative speeds are unbounded, and (b) arbitrary (Byzantine) failures of any client as well as up to one-third of the replicas (this is the theoretical lower bound [70]). These are called Byzantine-Fault-Tolerance SMR protocols, or simply *BFT* protocols, e.g., PBFT, QU, HQ and Zyzzyva [1, 27, 38, 67]. The ultimate goal of the designer of a BFT protocol is to exhibit comparable performance to a non-replicated server under "common" circumstances that are considered the most frequent in practice. The notion of "common" circumstance might depend on the application and underlying network, but it usually means network synchrony, rare failures, and sometimes also the absence of contention.

Not surprisingly, even under the same notion of "common" case, there is no "one size that fits all" BFT protocol. According to our own experience, the performance differences among the protocols can be heavily impacted by the actual network, the size of the messages, the very nature of the "common" case (e.g, contention or not); the actual number of clients, the total number of replicas as well as the cost of the cryptographic libraries being used. This echoes [87] which concluded for instance that *"PBFT [27] offers more predictable performance and scales better with payload size compared to Zyzzyva [67]; in contrast, Zyzzyva offers greater absolute throughput in wider-area, lossy networks"*. In fact, besides all BFT protocols mentioned above, there are good reasons to believe that we could design new protocols outperforming all others under specific circumstances. We do indeed present an example of a such protocol in this chapter.

To deploy a BFT solution, a system designer will hence certainly be tempted to adapt a state-of-the-art BFT protocol to the specific application/network setting, and possibly keep

adapting it whenever the setting changes. But this can rapidly turn into a nightmare. All protocol implementations we looked at involve around 20.000 lines of (non-trivial) C++ code, e.g., PBFT and Zyzzyva. Any change to an existing protocol, although algorithmically intuitive, is very painful. The changes of the protocol needed to optimize for the "common" case have sometimes strong impacts on the part of the protocol used in other cases (e.g., "view-change" in Zyzzyva). The only complete proof of a BFT protocol we knew of is that of PBFT and it involves 35 pages (even without using any formal language).[1] This difficulty, together with the impossibility of exhaustively testing distributed protocols [30] would rather plead for never changing a protocol that was widely tested, e.g., PBFT.

We propose in this chapter a way to have the cake and eat a big chunk of it. We present Abstract [58, 95] (*A*bortable Byzantine faulT-toleRant stAte maChine replicaTion): a new abstraction to reduce the development cost of BFT protocols. Following the divide-and-conquer principle, we view BFT protocols as a composition of instances of our abstraction, each instance targeted and optimized for specific system conditions. An instance of Abstract looks like BFT state machine replication, with one exception: it may sometimes *abort* a client's request.

The progress condition under which an Abstract instance should not abort is a generic parameter.[2] An extreme instance of Abstract is one that never aborts: this is exactly BFT. Interesting instances are "weaker" ones, in which an abort is allowed, e.g., if there is asynchrony or failures (or even contention). When such an instance aborts a client request, it returns a request history that is used by the client (proxy) to "recover" by switching to another instance of Abstract, e.g., one with a stronger progress condition. This new instance will commit subsequent requests until it itself aborts. This paves the path to *composability* and flexibility of BFT protocol design. Indeed, the composition of any two Abstract instances is *idempotent*, yielding yet another Abstract instance. Hence, and as we will illustrate in the chapter, the development (design, test, proof and implementation) of a BFT protocol boils down to:

- Developing individual Abstract instances. This is usually way much simpler than developing a full-fledged BFT protocol and allows for very effective schemes. A single Abstract instance can be crafted solely with its progress in mind, irrespective of other instances.

- Ensuring that a request is not aborted by all instances. This can be made very simple by reusing, as a black-box, an existing BFT protocol as one of the instances, without indulging into complex modifications.

To demonstrate the benefits of Abstract, we present two BFT protocols:

1. *AZyzzyva*, a protocol that illustrates the ability of Abstract to significantly ease the development of BFT protocols. *AZyzzyva* is the composition of two Abstract instances: (i) *ZLight*, which mimics Zyzzyva [67] when there are no asynchrony or failures, and (ii) *Backup*, which handles the periods with asynchrony/failures by reusing, as a black-box, a legacy BFT protocol. We leveraged PBFT which was widely tested, but could replace it with any BFT protocol. The code line count and proof size required to obtain *AZyzzyva* are, conservatively, less than $1/4$ than those of Zyzzyva. In some sense, had Abstract been identified several years ago, the designers of Zyzzyva would have had a much easier task devising a correct protocol exhibiting the performance they were seeking. Instead, they had to hack PBFT and, as a result, obtained a protocol that is way less stable than PBFT.

[1] It took Roberto De Prisco a PhD (MIT) to formally (using IOA) prove the correctness of a state machine protocol that does not even deal with malicious faults.

[2] Abstract can be viewed as a *virtual type*; each specification of the this progress condition defines a concrete type. These genericity ideas date back to the seminal chapter of Landin: *The Next 700 Programming Languages* (CACM, March 1966).

2. *Aliph*, a protocol that demonstrates the ability of Abstract to develop novel efficient BFT protocols. *Aliph* achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. *Aliph* uses, besides the *Backup* instance used in *AZyzzyva* (to handle the cases with asynchrony/failures), two new instances: (i) *Quorum*, targeted for system conditions that do not involve asynchrony/failures/contention, and (ii) *Chain*, targeted for high-contention conditions without failures/asynchrony. *Quorum* has a very low-latency (like e.g., [23, 44, 1]) and it makes *Aliph* the first BFT protocol to achieve a latency of only 2 message delays with as few as $3f + 1$ servers. *Chain* implements a pipeline message-pattern, and relies on a novel authentication technique. It makes *Aliph* the first BFT protocol with a number of MAC operations at the bottleneck server that tends to 1 in the absence of asynchrony/failures. This contradicts the claim that the lower bound is 2 [67]. Interestingly, each of *Quorum* and *Chain* could be developed independently and required less than 25% of the code needed to develop state-of-the-art BFT protocols.[3]

The rest of the chapter is organized as follows. Section 3.2 presents Abstract. After describing our system model in Section 3.3, we describe and evaluate our new BFT protocols: *AZyzzyva* in Section 3.4 and *Aliph* in Section 3.5. We presented related works in Section . Section 3.7 discusses the related work and concludes the chapter. For better readability, details are postponed to appendices. Appendix A contains the formal specification of Abstract. Appendix B contains protocol details with the correctness proofs given separately in Appendix C.

3.2 Abstract

We propose a new approach for the development of BFT protocols. We view a BFT protocol as a composition of instances of Abstract. Each instance is itself a protocol that commits clients' requests, like any state machine replication (SMR) scheme, except if certain conditions are not satisfied, in which case it can abort requests. These conditions, determined by the developer of the particular instance, capture the progress semantics of that instance. They might depend on the design goals and the environment in which a particular instance is to be deployed. Each instance can be developed, proved and tested independently, and this modularity comes from two crucial properties of Abstract:

1. *Switching* between instances is *idempotent*: the composition of two Abstract instances yields yet another Abstract instance.

2. BFT is nothing but a special Abstract instance — *one that never aborts*.

A correct implementation of an Abstract instance always preserves BFT safety — this extends to any composition thereof. The designer of a BFT protocol only has to make sure that: a) individual Abstract implementations are correct, *irrespectively of each other*, and b) the composition of the chosen instances is live: i.e. that every request will eventually be committed. We exemplify this later, in Sections 3.4 and 3.5. In the following, we highlight the main characteristics of Abstract. For better readability, precise specification of Abstract and our theorem on Abstract switching idempotency are postponed to Appendix A.

3.2.1 Switching

Every Abstract instance has a unique identifier (instance number) i. When an instance i commits a request, i returns a state-machine reply to the invoking client. Like any SMR

[3]Our code counts are in fact conservative since they do not discount for the libraries shared between *ZLight*, *Quorum* and *Chain*, which amount to 10% of a state-of-the-art BFT protocol.

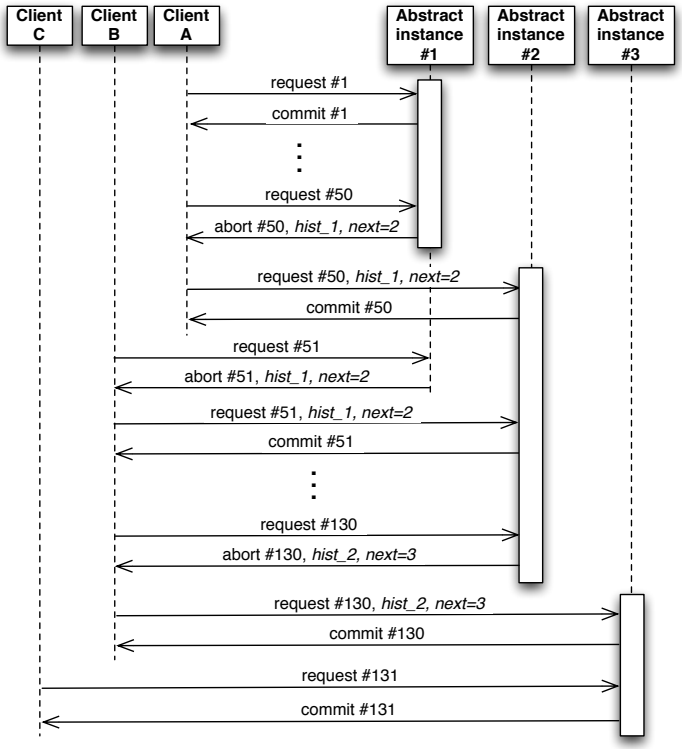


Figure 3.1: Abstract operating principle.

scheme, i establishes a total order on all committed requests according to which the reply is computed for the client. If, however, i aborts a request, it returns to the client a digest of the history of requests h that were committed by i (possibly along with some uncommitted requests); this is called an *abort history*. In addition, i returns to the client the identifier of the next instance ($next(i)$) which should be invoked by the client: $next$ is the same function across all abort indications of instance i, and we say instance i *switches* to instance $next(i)$. In the context of this chapter, we consider $next$ to be a pre-determined function (e.g., known to servers implementing a given Abstract instance); we talk about *deterministic* or *static* switching. However, this is not required by our specification; $next(i)$ can be computed "on-fly" by the Abstract implementation (e.g., depending on the current workload, or possible failures or asynchrony) as long as $next$ remains a function. In this case, we talk about *dynamic switching*; this is out of the scope of this chapter.

The client uses abort history h of i to invoke $next(i)$; in the context of $next(i)$, h is called an *init* history. Roughly speaking, $next(i)$ is initialized with an init history, before it starts committing/aborting clients' requests. The initialization serves to transfer to instance $next(i)$ the information about the requests committed within instance i, in order to preserve total order among committed requests across the two instances.

Once i aborts some request and switches to $next(i)$, i cannot commit any subsequently invoked request. We impose *switching monotonicity*: for all i, $next(i) > i$. Consequently, Abstract instance i that fails to commit a request is abandoned and all clients go from there on to the next instance, never re-invoking i.

3.2.2 Illustration

Figure 3.1 depicts a possible run of a BFT system built using Abstract. To preserve consistency, Abstract properties ensure that, at any point in time, only one Abstract instance, called *active*, may commit requests. Client A starts sending requests to the first Abstract instance. The latter commits requests #1 to #49 and aborts request #50, becoming inactive. Abstract appends to the abort indication an (unforgeable) history ($hist_1$) and the information about the next Abstract instance to be used ($next = 2$). Client A sends to the new Abstract instance both its uncommitted request ($\#50$) and the history returned by the first Abstract instance. Instance $\#2$ gets initialized with the given history and executes request $\#50$. Later on, client B sends request $\#51$ to the first Abstract instance. The latter returns an abort indication with a possibly different history than the one returned to client A (yet both histories must contain previously committed requests #1 to #49). Client B subsequently sends request $\#51$ together with the history to the second abstract instance. The latter being already initialized, it simply ignores the history and executes request $\#51$. The second abstract instance then executes the subsequent requests up to request $\#130$ which it aborts. Client B uses the history returned by the second abstract instance to initialize the third abstract instance. The latter executes request $\#130$. Finally, Client C, sends request $\#131$ to the third instance, that executes it. Note that unlike Client B, Client C directly accesses the currently active instance. This is possible if Client C knows which instance is active, or if all three Abstract instances are implemented over the same set of replicas: replicas can then, for example, 'tunnel' the request to the active instance.

3.2.3 A View-Change Perspective

In some sense, an Abstract instance number can be seen as a view number, e.g., in PBFT [27].[4] Like in existing BFT protocols, which merely reiterate the exact same subprotocol across the views (possibly changing the server acting as *leader*), the same Abstract

[4]The opposite however does not hold, since multiple views of a given BFT protocol can be captured within a single Abstract instance.

implementations can be re-used (with increasing instance numbers). However, unlike existing BFT protocols, Abstract compositions *allow entire sub-protocols to be changed on a 'view-change'* (i.e., during switching).

3.2.4 Misbehaving Clients

Clients that fail to comply with the switching mechanism (e.g., by inventing/forging an init history) cannot violate the Abstract specification. Indeed, to be considered valid, an init history of $next(i)$ must be previously returned by the preceding Abstract i as an abort history. To enforce this causality, in practice, our Abstract compositions (see Sec. 3.4 and Sec. 3.5) rely on unforgeable digital signatures to authenticate abort histories in the presence of potentially Byzantine clients. View-change mechanisms employed in existing BFT protocols [27, 67], have similar requirements: they exchange digitally signed messages.

3.3 System Model

We assume a message-passing distributed system using a fully connected network among processes: clients and servers. The links between processes are asynchronous and unreliable: messages may be delayed or dropped (we speak of link failures). However, we assume fair-loss links: a message sent an infinite number of times between two correct processes will be eventually received. Processes are Byzantine fault-prone; processes that do not fail are said to be correct. A process is called *benign* if it is correct or if it fails by simply crashing. In our algorithms, we assume that any number of clients and up to f out of $3f + 1$ servers can be Byzantine. We assume a strong adversary that can coordinate faulty nodes; however, we assume that the adversary cannot violate cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), and signatures.

We further assume that during *synchronous* periods (i.e., when there are no link failures) any message m sent between two correct processes is delivered within a bounded delay Δ (known to sender and receiver) if the sender retransmits m until it is delivered.

Finally, we declare *contention* in an Abstract instance whenever there are two concurrent requests such that both requests are invoked but not yet committed/aborted.

3.4 Putting Abstract to Work: AZyzzyva

We illustrate how Abstract significantly eases the design, implementation, and proof of BFT protocols with *AZyzzyva*. This is a full fledged BFT protocol that mimics Zyzzyva [67] in its "common case" (i.e., when there are no link or server failures). In "other cases" we rely on *Backup*, an Abstract implementation with strong progress guarantees that can be implemented on top of *any* existing BFT protocol. In our implementation, we chose PBFT [27] for it has been extensively tested and proved correct. We chose to mimic Zyzzyva, for it is known to be efficient, yet very difficult to implement [35]. Using Abstract, we had to write, prove and test less than 24% of the Zyzzyva code to obtain *AZyzzyva*.

In the "common case", Zyzzyva executes the fast speculative path depicted in Figure 3.2. A client sends a request to a designated server, called *primary* (r_1 in Fig. 3.2). The primary appends a sequence number to the request and broadcasts the request to all replicas. Each replica speculatively executes the request and sends a reply to the client. All messages in the above sequence are authenticated using MACs rather than (more expensive) digital signatures. The client commits the request if it receives the same reply from all $3f + 1$ replicas. Otherwise, Zyzzyva executes a second phase that aims at handling the case with link/server/client failures ("worst-case"). Roughly, this phase (that *AZyzzyva* avoids to mimic) consists of considerable modifications to PBFT [27], which arise from the "profound

effects" [67], that the Zyzzyva "common-case" optimizations have on its "worst-case". The second phase is so complex that, as confessed by the authors themselves [35], it is not entirely implemented in the current Zyzzyva prototype. In fact, when this second phase is stressed, due to its complexity and the inherent bugs that it contains, the throughput of Zyzzyva drops to 0.

In the following, we describe how we build *AZyzzyva*, assess the qualitative benefit of using Abstract and discuss the performance of *AZyzzyva*.

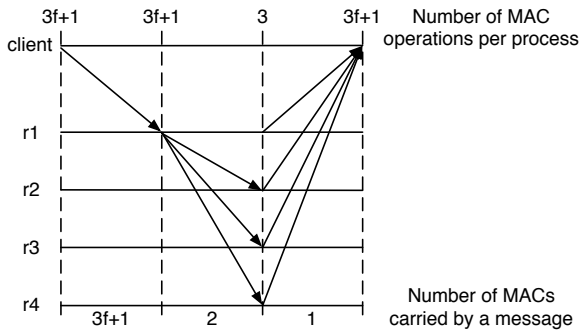


Figure 3.2: Communication pattern of *ZLight*.

3.4.1 Protocol Overview

Our goal when building *AZyzzyva* using Abstract is to show that we can completely *separate the concerns* of handling the "common-case" and the "worst-case". We thus use two different Abstract implementations: *ZLight* and *Backup*. Roughly, *ZLight* is a Abstract that guarantees progress in the Zyzzyva "common-case". On the other hand, *Backup* is an Abstract with strong progress: it guarantees to commit an exact certain number of requests k (k is itself configurable) before it starts aborting.

We then simply construct *AZyzzyva* such that every odd (resp., even) Abstract instance is *ZLight* (resp., *Backup*). *ZLight* is first executed. When it aborts, it switches to *Backup*, which commits the next k requests. *Backup* then aborts subsequent requests and switches to (a new instance of) *ZLight*, and so on.

Note that *ZLight* uses a lightweight checkpointing protocol (shared with *Aliph*'s *Quorum* and *Chain*, Sec. 3.5) triggered every 128 messages to truncate histories (see Sec. B.6.1).

In the following, we briefly describe *ZLight* and *Backup*. Details are postponed to Appendix B, whereas correctness proofs can be found in Appendix C.

3.4.2 ZLight

ZLight implements Abstract with the following progress property which reflects Zyzzyva "common case": it commits requests when (a) there are no server or link failures, and (b) no client is Byzantine (simple client crash failures are tolerated). When this property holds, *ZLight* implements *Zyzzyva* "common-case" pattern (Fig. 3.2), described earlier. Outside the "common-case", when a client does not receive $3f + 1$ consistent replies, the client sends a PANIC message to replicas. Upon reception of this message, replicas stop executing requests and send back a signed message containing their history (replicas will now send the same abort message for all subsequent requests). When the client receives $2f + 1$ signed messages containing replica histories, it can generate an abort history and switch to *Backup*. Client generates abort history ah such that $ah[j]$ equals the value that appears at position $j \geq 1$ of

$f + 1$ different replica histories (the details of the panic and switching mechanisms are in the appendix, in Sec. B.5 and B.6, respectively).

3.4.3 Backup

Backup is an Abstract implementation with a progress property that guarantees exactly $k \geq 1$ requests to be committed, where k is a generic parameter (we explain our configuration for k at the end of this section). We employ *Backup* in *AZyzzyva* (and *Aliph*) to ensure progress outside "common-cases" (e.g., under replica failures).

We implemented *Backup* as a very thin wrapper (around 600 lines of C++ code) that can be put around *any existing* BFT protocol. In our C/C++ implementations, *Backup* is implemented over PBFT [27], for PBFT is the most extensively tested BFT protocol and it is proven correct. Other existing BFT protocols that provide robust performance under failures, like Aardvark [35], are also very good candidates for the *Backup* basis (unfortunately, the code of Aardvark is not yet publicly available).

To implement *Backup*, we exploit the fact that any BFT can totally order requests submitted to it and implement any functionality on top of this total order. In our case, *Backup* is precisely this functionality. *Backup* works as follows: it ignores all the requests delivered by the underlying BFT protocol until it receives a request containing a valid init history, i.e. an unforgeable abort history generated by the preceding Abstract (*ZLight* in the case of *AZyzzyva*). At this point, *Backup* sets its state by executing all the requests contained in a valid init history it received. Then, it simply executes the first k requests ordered by BFT (neglecting subsequent init histories) and commits these requests. After committing the kth request, *Backup* aborts all subsequent requests returning the signed sequence of executed requests as the abort history (replica digital signature functionality assumed here is readily present in all existing BFT protocols we know of).

The parameter k is generic and is an integral part of the *Backup* progress guarantees. Our default configuration increases k exponentially, with every new instance of *Backup*. This ensures the liveness of the composition, which might not be the case with, say, a fixed k in a corner case with very slow clients[5]. More importantly, in the case of failures, we actually do want to have a *Backup* instance remaining active for long enough, since *Backup* is precisely targeted to handle failures. On the other hand, to reduce the impact of transient link failures, which can drive k to high values and thus confine clients to *Backup* for a long time after the transient failure disappears, we flatten the exponential curve for k to maintain $k = 1$ during some targeted outage time[6]. In our implementation, we also periodically reset k. Dynamically adapting k to fit the system conditions is appealing but requires further studies and is out of the scope of this chapter.

3.4.4 Qualitative Assessment

In evaluating the effort of building *AZyzzyva*, we focus on the cost of *ZLight*. Indeed, *Backup*, for which the additional effort is small (around 600 lines of C++ code), can be reused for other BFT protocols in our framework. For instance, we use *Backup* in our *Aliph* protocol as well (Sec. 3.5).

Table 3.1 compares the number of pages of pseudo-code, pages of proofs and lines of code of Zyzzyva and *ZLight*. The comparison in terms of lines of code is fair, since *Zyzzyva* and

[5]In short, k requests committed by a single *Backup* instance i might all be invoked by the same, fast client. A slow client can then get its request aborted by i. The same can happen with a subsequent *Backup* instance, etc. This issue can be avoided by exponentially increasing k (for any realistic load that does not increase faster than exponentially) or by having the replicas across different Abstract instances share a client input buffer.

[6]For example, using $k = \lceil C * 2^m \rceil$, where m is incremented with every new Abstract instance, with the rough average time of $50ms$ for switching between 2 consecutive *Backup* instances in *AZyzzyva*, we can maintain $k = 1$ during $10s$ outages with $C = 2^{-200}$.

all protocols presented in this chapter use the same code base (inherited from PBFT [27]). Notice that, all implementations in the PBFT code base, share about 7500 lines of code implementing cryptographic functions, data structures (e.g. maps, sets), etc. We do not count these lines, which we packaged in a separate library. The code line comparison shows that to build *ZLight* we needed less than 24% of the *Zyzzyva* line count (14339 lines). This is, however, conservative since we needed only about 14% to implement *ZLight* "common case"; the remaining 10% (1391 lines) are due to panicking and checkpointing mechanisms, which are all shared among *ZLight*, *Quorum* and *Chain* (the latter two are used in *Aliph*, Sec. 3.5). The difference in the *ZLight* vs. *Zyzzyva* code size is that *ZLight* aborts as soon as the system conditions fall outside the "common-case" (in which case *AZyzzyva* shifts the load to *Backup*). Hence, we avoid the "common-case"/"worst-case" code dependency that plagued Zyzzyva.

Using the same syntax as the one used in the original Zyzzyva chapter [67], *ZLight* requires approximately half a page of pseudo-code, its plain-english proof requires about 1 page (see Sec. C.1). In comparison, the pseudo-code of Zyzzyva (without checkpointing) requires 4.5 pages, making it about 9 times bigger than that of *ZLight*. Due to the complexity of Zyzzyva, the authors first presented a version using signatures and then explained how to modify it to use MACs. The correctness proof of the *Zyzzyva* signature version requires 4 (double-column) pages, whereas the proof for the MAC version is only sketched.

	Zyzzyva	*ZLight*
Pages of pseudo-code	4,5	0,5
Pages of proofs	> 4	1
Lines of code	14339	3358

Table 3.1: Complexity comparison of Zyzzyva and *ZLight*.

3.4.5 Performance Evaluation

We have compared the performance of AZyzzyva and Zyzzyva in the "common-case", using the benchmarks described in Section 3.5.2. Not surprisingly, AZyzzyva and Zyzzyva have the exact same performance in this case. In this section, we do thus focus on the cost induced by Abstract switching mechanism when the operating conditions are outside the common-case (and *ZLight* aborts a request). We could not compare against Zyzzyva. Indeed, as explained above, it has bugs in the second phase in charge of handling faults, which makes its impossible to evaluate the current prototype outside the "common-case".

To assess the switching cost, we perform the following experiments: we feed the request history of *ZLight* with r requests of size $1kB$. We then issue 10000 successive requests. To isolate the cost of the switching mechanism, we do not execute the *ZLight* common case; the measured time comprises the time required (1) by the client to send a PANIC message to *ZLight* replicas, (2) by the replicas to generate and send a signed message containing their history, (3) by the client to invoke Backup with the abort/init history, and (4) by the (next) client to get the abort history from Backup and initialize the next *ZLight* instance. Note that we deactivate the functions in charge of updating the history of *ZLight*, so that we ensure that for each aborted request, the history contains r requests. We reproduced each experiment three times and observed a negligible variance.

Figure 3.3 shows the switching time (in ms) as a function of the history size when the number of tolerated faults equals 1. As mentioned above, *ZLight* uses a checkpointing mechanism triggered every 128 requests. To account for requests that might be received by servers while they are performing a checkpoint, we assume that the history size can grow up to 250 requests. We plot two different curves: one corresponds to the case when replicas do not

miss any request. The other one corresponds to the case when replicas miss requests. More precisely, we assess the performance when 30% of the requests are absent from the history of at least one replica. Not surprisingly, we observe that the switching cost increases with the history size and that it is slightly higher in the case when replicas miss requests (as replicas need to fetch the requests they miss). Interestingly, we see that the switching cost is very reasonable. It ranges between $19.7ms$ and $29.2ms$, which is low provided faults are supposed to be rare in the environment for which Zyzzyva has been devised.

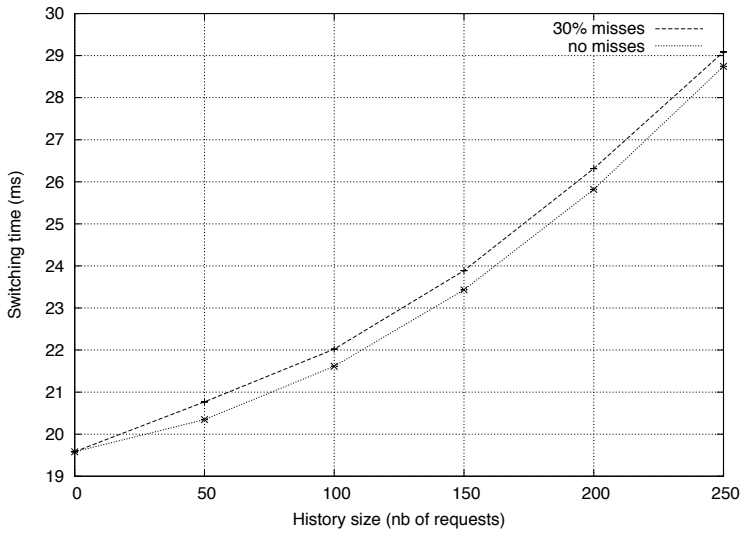


Figure 3.3: Switching time in function of history size and percentage of missing requests in replica histories.

3.5 Putting Abstract to Really Work: Aliph

In this section, we demonstrate how we can build novel, very efficient BFT protocols, using Abstract. Our new protocol, called *Aliph*, achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. The development of *Aliph* consisted in building two new instances of Abstract, each requiring less than 25% of the code of state-of-the-art protocols, and reusing *Backup* (Sec. 3.4.3). In the following, we first describe *Aliph* and then we evaluate its performance.

3.5.1 Protocol Overview

The characteristics of *Aliph* are summarized in Table 3.2, considering the metrics of [67]. In short, *Aliph* is the first optimally resilient protocol that achieves a latency of 2 one-way message delays when there is no contention. It is also the first protocol for which the number of MAC operations at the bottleneck replica tends to 1 (under high contention when batching of messages is enabled): 50% less than required by state-of-the-art protocols.

Aliph uses three Abstract implementations: *Backup* (introduced in Sec. 4.3), *Quorum* and *Chain* (both described below). A *Quorum* instance commits requests as long as there are no: (a) server/link failures, (b) client Byzantine failures, and (c) contention. *Quorum* implements a very simple communication pattern and gives *Aliph* the low latency flavor when its progress conditions are satisfied. On the other hand, *Chain* provides exactly the same progress guarantees as *ZLight* (Sec. 3.4.2), i.e., it commits requests as long as there are no

	PBFT	Q/U	HQ	Zyzzyva	Aliph
Number of replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1**
Throughput (MAC ops at bottleneck server)	$2+\frac{8f}{b}$	2+4f	2+4f	$2+\frac{3f}{b}$	**1+**$\frac{f+1}{b}$
Latency (1-way messages in the critical path)	4	**2**	4	3	**2**

Table 3.2: Characteristics of state-of-the-art BFT protocols. Row 1 is the number of replicas. Row 2 is the throughput in terms of number of MAC operations at the bottleneck replica (assuming batches of b requests). Row 3 is the latency in terms of number of 1-way messages in the critical path. Bold entries denote protocols with the lowest known cost.

server/link failures or Byzantine clients. *Chain* implements a pipeline pattern and, as we show below, allows *Aliph* to achieve better peak throughput than all existing protocols. Both *Quorum* and *Chain* share the panicking mechanism with *ZLight*, which is invoked by the client when it fails to commit the request.

Aliph uses the following static switching ordering to orchestrate its underlying protocols: *Quorum-Chain-Backup-Quorum-Chain-Backup*$-etc$. Initially, *Quorum* is active. As soon as it aborts (e.g., due to contention), it switches to *Chain*. *Chain* commits requests until it aborts (e.g., due to asynchrony). *Aliph* then switches to *Backup*, which executes k requests (see Sec. 3.4.3). When *Backup* commits k requests, it aborts, switches back to *Quorum*, and so on.

In the following, we first describe *Quorum* (Sec. 3.5.1.1) and *Chain* (Sec. 3.5.1.2) (full details and correctness proofs can be found in Appendix B and C, respectively). Then, we discuss some system-level optimizations of *Aliph* (Sec. 3.5.1.3).

3.5.1.1 Quorum

Quorum implements a very simple communication pattern (see Fig. 3.4); it requires only one round-trip of message exchange between a client and replicas to commit a request. Namely, the client sends the request to all replicas that speculatively execute it and send a reply to the client. As in *ZLight*, replies sent by replicas contain a digest of their history. The client checks that the histories sent by the $3f + 1$ replicas match. If that is not the case, or if the client does not receive $3f + 1$ replies, the client invokes a panicking mechanism. This is the same as in *ZLight* (Sec. 3.4.2): (i) the client sends a PANIC message to replicas, (ii) replicas stop executing requests on reception of a PANIC message, (iii) replicas send back a signed message containing their history. The client collects $2f + 1$ signed messages containing replica histories and generates an abort history. Note that, unlike *ZLight*, *Quorum* does not tolerate contention: concurrent requests can be executed in different orders by different replicas. This is not the case in *ZLight*, as requests are ordered by the primary.

The implementation of *Quorum* is very simple. It requires 3200 lines of C code (including panicking and checkpoint libraries). *Quorum* makes *Aliph* the first BFT protocol to achieve a latency of 2 one-way message delays, while only requiring $3f + 1$ replicas (Q/U [1] has the same latency but requires $5f + 1$ replicas). Given its simplicity and efficiency, it might seem surprising not to have seen it published earlier. We believe that Abstract made that possible because we could focus on weaker (and hence easier to implement) Abstract specifications, without caring about (numerous) difficulties outside the *Quorum* "common-case".

3.5.1.2 Chain

Chain organizes replicas in a pipeline (see Fig. 3.5). All replicas know the fixed ordering of replica IDs (called *chain order*); the first (resp., last) replica is called the *head* (resp., the *tail*). Without loss of generality we assume an ascending ordering by replica IDs, where the head (resp., tail) is replica r_1 (resp., r_{3f+1}).

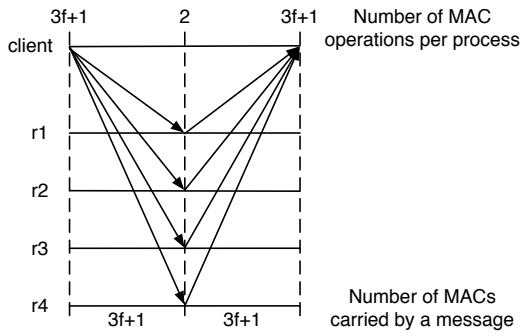


Figure 3.4: Communication pattern of *Quorum*.

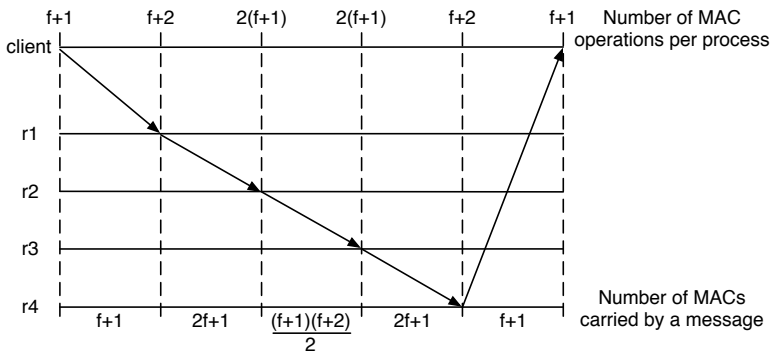


Figure 3.5: Communication pattern of *Chain*.

In *Chain*, a client on invoking a request, sends the request req to the head, who assigns sequence numbers to requests. Then, each replica r_i forwards the message to its *successor* $\overrightarrow{r_i}$, where $\overrightarrow{r_i} = r_{i+1}$. The exception is the tail whose successor is the client: upon receiving the message, the tail sends the reply to the client. Similarly, replica r_i in *Chain* accepts a message only if sent by its predecessor $\overleftarrow{r_i}$, where $\overleftarrow{r_i} = r_{i-1}$; the exception is the head, which accepts requests only from the client.

The behavior of *Chain*, as described so far, is very similar to the protocol described in [93]. That protocol, however, did only tolerate crash faults. We tolerate Byzantine failures by ensuring: (1) that the content of a message is not modified by a malicious replica before being forwarded, (2) that no replica in the chain is bypassed, and (3) that the reply sent by the tail is correct. To provide those guarantees, our *Chain* relies on a novel authentication method we call *chain authenticators (CAs)*. CAs are lightweight MAC authenticators, requiring processes to generate (at most) $f + 1$ MACs (in contrast to $3f + 1$ in traditional authenticators). CAs guarantee that, if a client commits request req, every correct replica executed req. CAs, along with the inherent throughput advantages of a pipeline pattern, are key to *Chain*'s dramatic throughput improvements over other BFT protocols. We describe below how CAs are used in *Chain*.

Replicas and clients generate CAs in order to authenticate the messages they send. Each CA contains MACs for a set of processes called *successor* set. The successor set of clients consists of the $f+1$ first replicas in the chain. The successor set of a replica r_i depends on its position i: (a) for the first $2f$ replicas, the successor set comprises the next $f + 1$ replicas in the chain, whereas (b) for other replicas ($i > 2f$), the successor set comprises all subsequent replicas in the chain, as well as the client. Dually, when process p receives a message m it *verifies* m, i.e., it checks whether m contains a correct MAC from the processes from p's *predecessor set* (a set of processes q such that p is in q's successor set). For instance, process p_1 verifies that the message contains a valid MAC from process p_0 and the client, whereas the client verifies that the reply it gets contains a valid MAC from the last $f + 1$ replicas in the chain. Finally, to make sure that the reply sent by the tail is correct, the f processes that precede the tail in the chain append a digest of the response to the message.

When the client receives a correct reply, it commits it. On the other hand, when the reply is not correct, or when it does not receive any reply (e.g., due to the Byzantine tail which discards the request), the client broadcasts a PANIC message to replicas. As in *ZLight* and *Quorum*, when replicas receive a PANIC message, they stop executing requests and send back a signed message containing their history. The client collects $2f + 1$ signed messages containing replica histories and generates an abort history.

Chain's implementation requires 3300 lines of code (with panic and checkpoint libraries). Moreover, it is the first protocol in which the number of MAC operations at the bottleneck replica tends to 1. This comes from the fact that, under contention, the head of the chain can batch requests. Head and tail do thus need to read (resp. write) a MAC from (resp. to) the client, and write (resp. read) $f + 1$ MACs for a batch of requests. Thus for a single request, head and tail perform $1 + \frac{f+1}{b}$ MAC operations. Note that all other replicas process requests in batch, and have thus a lower number of MAC operations per request. State-of-the-art protocols [67, 27] do all require at least 2 MAC operations at the bottleneck server (with the same assumption on batching). The reason why this number tends to 1 in *Chain* can be intuitively explained by the fact that these are two distinct replicas that receive the request (the head) and send the reply (the tail).

3.5.1.3 Optimizations

When a *Chain* instance is executing in *Aliph*, it commits requests as long as there are no server or link failures. In the *Aliph* implementation we benchmark in the evaluation, we slightly modified the progress property of *Chain* so that it aborts requests as soon as

replicas detect that there is no contention (i.e. there is only one active client since at least $2s$). Moreover, *Chain* replicas add an information in their abort history to specify that they aborted because of the lack of contention. We slightly modified *Backup*, so that in such case, it only executes one request and aborts. Consequently, *Aliph* switches to *Quorum*, which is very efficient when there is no contention. Finally, in *Chain* and *Quorum* we use the same checkpoint protocol as in *ZLight*.

3.5.2 Evaluation

This section evaluates the performance of *Aliph*. For lack of space, we focus on experiments without failures (of processes or links), since we compare to protocols that are known to perform well in the common-case — PBFT [27], Q/U [1] and Zyzzyva [67].

We first study latency, throughput, and fault scalability using Castro's microbenchmarks [27, 67], varying the number of clients. Clients invoke requests in closed-loop (meaning that a client does not invoke a new request before it commits a previous one). In the x/y microbenchmark, clients send xkB requests and receive ykB replies. We also perform an experiment in which the input load dynamically varies.

We evaluate PBFT and Zyzzyva because the former is considered the "baseline" for practical BFT implementations, whereas the latter is considered state-of-the-art. Moreover, Zyzzyva systematically outperforms HQ [67]; hence, we do not evaluate HQ. Finally, we benchmark Q/U as it is known to provide better latency than Zyzzyva under certain condition. Note that Q/U requires $5f + 1$ servers, whereas other protocols we benchmark only require $3f + 1$ servers.

PBFT and Zyzzyva implement two optimizations: request batching by the primary, and client multicast (in which clients send requests directly to all the servers and the primary only sends ordering messages). All measurements of PBFT are performed with batching enabled as it always improves performance. This is not the case in Zyzzyva. Therefore, we assess Zyzzyva with or without batching depending on the experiment. As for the client multicast optimization, we show results for both configurations every time we observe an interesting behavior.

PBFT code base underlies both Zyzzyva and *Aliph*. To ensure that the comparison with Q/U is fair, we evaluate its simple best-case implementation described in [67].

We ran all our experiments on a cluster of 17 identical machines, each equipped with a 1.66GHz bi-processor and 2GB of RAM. Machines run the Linux 2.6.18 kernel and are connected using a Gigabit ethernet switch.

3.5.2.1 Latency

	0/0 benchmark			**4/0 benchmark**			**0/4 benchmark**		
	f=1	**f=2**	**f=3**	**f=1**	**f=2**	**f=3**	**f=1**	**f=2**	**f=3**
Q/U	8%	14,9%	33,1%	6,5%	13,6%	22,3%	4,7%	20,2%	26%
Zyzzyva	31,6%	31,2%	34,5%	27,7%	26,7%	15,6%	24,3%	26%	15,6%
PBFT	49,1%	48,8%	44,5%	36,6%	38,4%	26%	37,6%	38,2%	29%

Table 3.3: Latency improvement of *Aliph* for the 0/0, 4/0, and 0/4 benchmarks.

We first assess the latency in a system without contention, with a single client issuing requests. The results for all microbenchmarks (0/0, 0/4 and 4/0) are summarized in Table 3.3 demonstrating the latency improvement of *Aliph* over Q/U, PBFT, and Zyzzyva. We show results for a maximal number of server failures f ranging from 1 to 3. Our results show that *Aliph* consistently outperforms other protocols, since *Quorum* is active when there is

no contention. These results confirm the theoretical expectations (see Table 3.2, Sec. 3.5.1). The results show that Q/U also achieves a good latency with $f = 1$, as Q/U and *Quorum* use the same communication pattern. Nevertheless, when f increases, performance of Q/U decreases significantly. The reason is that Q/U requires $5f + 1$ replicas and both clients and servers perform additional MAC computations compared to *Quorum*. Moreover, the significant improvement of *Aliph* over Zyzzyva (31% at $f = 1$) can be easily explained by the fact that Zyzzyva requires 3-one-way message delays in the common case, whereas *Aliph* (*Quorum*) only requires 2-one-way message delays.

3.5.2.2 Throughput

In this section, we present results obtained running the 0/0, 0/4, and 4/0 microbenchmarks under contention. We do not present the results for Q/U since it is known to perform poorly under contention. Notice that in all the experiments presented in this section, *Chain* is active in *Aliph*. The reason is that, due to contention, there is always a point in time when a request sent to *Quorum* reaches replicas in a different order, which results in a switch to *Chain*. As there are no failures in the experiments presented in this section, *Chain* executes all the subsequent requests.

Our results show that *Aliph* consistently and significantly outperforms other protocols starting from a certain number of clients that depends on the benchmark. Below this threshold, Zyzzyva achieves higher throughput than other protocols.

0/0 benchmark. Figure 3.6 plots the throughput achieved in the 0/0 benchmark by various protocols when $f = 1$. We ran Zyzzyva with and without batching. For PBFT, we present only the results with batching, since they are substantially better than results without batching. We observe that Zyzzyva with batching performs better than PBFT, which itself achieves higher peak throughput than Zyzzyva without batching (this is consistent with the results of [67, 87]).

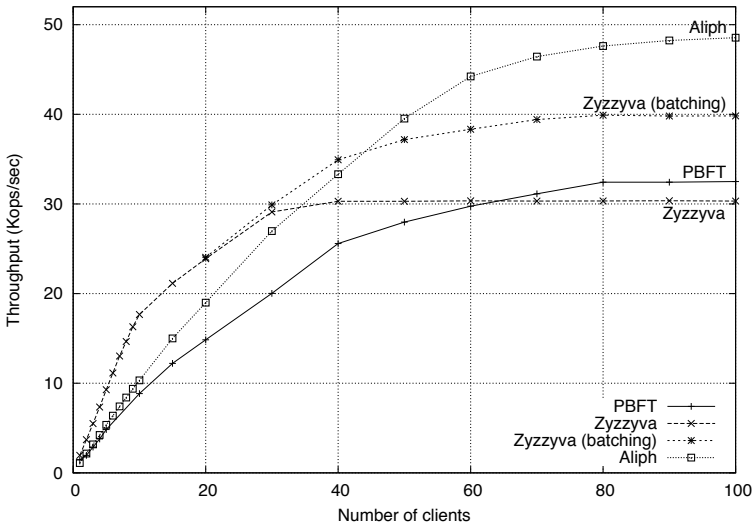


Figure 3.6: Throughput for the 0/0 benchmark (f=1).

Moreover, Figure 3.6 shows that with up to 40 clients, Zyzzyva achieves the best throughput. With more than 40 clients, *Aliph* starts to outperform Zyzzyva. The peak throughput achieved by *Aliph* is 21% higher than that of Zyzzyva. The reason is that *Aliph* executes *Chain*, which uses a pipeline pattern to disseminate requests. This pipeline pattern brings

two benefits: reduced number of MAC operations at the bottleneck server, and better network usage: servers send/receive messages to/from a single server.

Nevertheless, the *Chain* is efficient only if its pipeline is fed, i.e. the link between any server and its successor in the chain must be saturated. There are two ways to feed the pipeline: using large messages (see the next benchmark), or a large number of small messages (this is the case of 0/0 benchmark). Moreover, as in the microbenchmarks clients invoke requests in closed-loop, it is necessary to have a large number of clients to issue a large number of requests. This explains why *Aliph* starts outperforming Zyzzyva only with more than 40 clients.

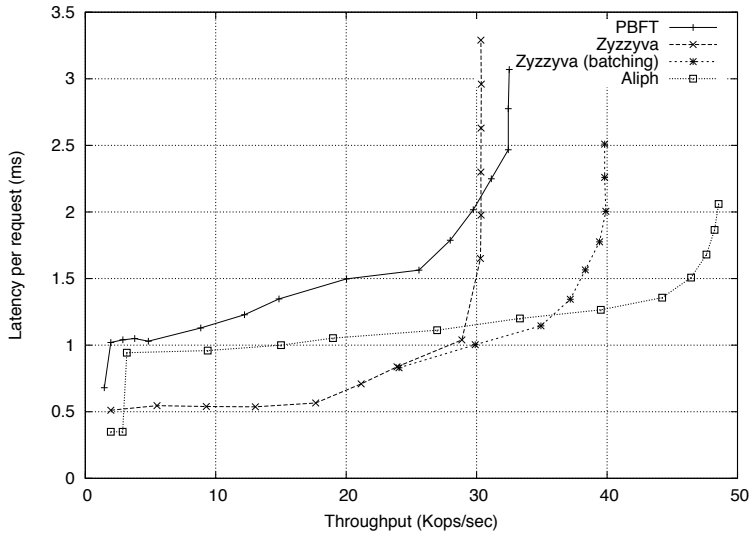


Figure 3.7: Response-time vs. throughput for the 0/0 benchmark (f=1).

Figure 3.7 plots the response-time of Zyzzyva (with and without batching), PBFT and *Aliph* as a function of the achieved throughput. We observe that *Aliph* achieves consistently lower response-time than PBFT. This stems from the fact that the message pattern of PBFT is a very complex one, which increases the response time and limits the throughput of PBFT. Moreover, up to the throughput of 37Kops/sec, *Aliph* has a slightly higher response-time than Zyzzyva. The reason is the pipeline pattern of *Chain* that results in a higher response time for low to medium throughput, which stays reasonable nevertheless. Moreover, *Aliph* scales better than Zyzzyva: from 37Kops/sec, it achieves lower response time, since the messages are processed faster due to the higher throughput ensured by *Chain*. Hence, messages spend less time in waiting queues. Finally, we observe that for very low throughput, *Aliph* has lower response time than Zyzzyva. This comes from the fact that *Aliph* uses *Quorum* when there is no contention, which significantly improves the response-time of the protocol.

0/4 benchmark. Figure 3.8 shows the throughput of the various protocols for the 0/4 microbenchmark when $f = 1$. PBFT and Zyzzyva are using the client multicast optimization. We observe that with up to 15 clients, Zyzzyva outperforms other protocols. Starting from 20 clients, *Aliph* outperforms PBFT and Zyzzyva. Nevertheless, the gain in peak throughput (7,7% over PBFT and 9,8% over Zyzzyva) is lower than the gain we had with the 0/0 microbenchmark. This can be explained by the fact that the dominating cost is now sending replies to clients, partly masking the effect of request processing and request/sequence number forwarding. In all protocols, there is only one server sending a full reply to the client (other servers send only a digest of the reply). We were expecting PBFT and Zyzzyva to out-

perform *Aliph* (which executes *Chain* when there is load), since the server that sends a full reply in PBFT and Zyzzyva changes on a per-request basis. Nevertheless, this is not the case. We again attribute this result to the fact that *Chain* uses a pipeline pattern: the last process in the chain replies to clients at the throughput of about 391MB/sec.

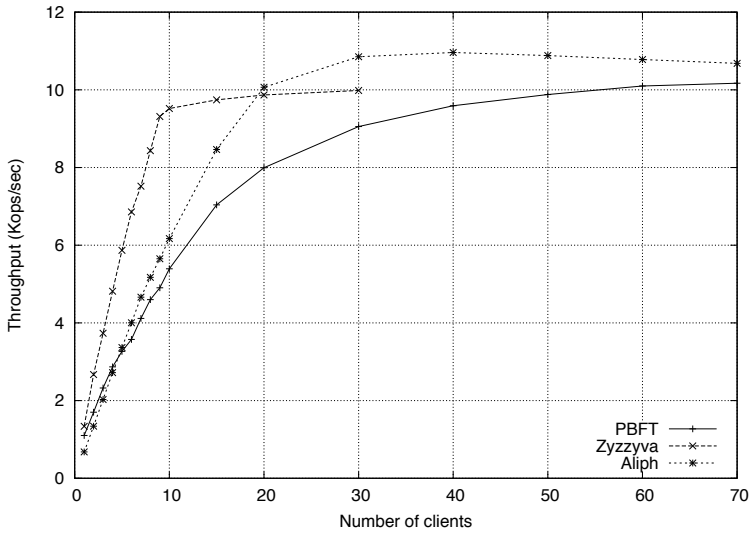


Figure 3.8: Throughput for the 0/4 benchmark (f=1).

4/0 benchmark. Figure 3.9 shows the results of *Aliph*, PBFT and Zyzzyva for the 4/0 microbenchmark with $f = 1$. Notice the logarithmic scale for the X axis, that we use to better highlight the behavior of various protocols with small numbers of clients. For PBFT and Zyzzyva, we plot curves both with and without client multicast optimization. The graph shows that with up to 3 clients, Zyzzyva outperforms other protocols. With more than 3 clients, *Aliph* significantly outperforms other protocols. Its peak throughput is about 360% higher than that of Zyzzyva. The reason why *Aliph* is very efficient under high load and when requests are large was explained earlier in the context of the 0/0 benchmark.

Notice also the interesting drop in the performance of Zyzzyva and PBFT when client multicast optimization is used (Fig. 3.9). This is to be contrasted with the case when the primary forwards requests, where the performance of PBFT and Zyzzyva remain almost constant after the peak throughput has been reached. These results may seem surprising given that [67, 27] recommend to use the client multicast optimization when requests are large, in order to spare the primary of costly operations request forwarding. Nevertheless, these results can be explained by the fact that simultaneous multicasts of large messages by different clients result in collisions and buffer overflows, thus requiring many message retransmissions[7] (there is no flow control in UDP). This explains why enabling the concurrent client multicasts drastically reduces performance. On the other hand, when the primary forwards messages, there are fewer collisions, which explains the better performance we observe.

Impact of the request size. In this experiment we study how protocols are impacted by the size of requests. Figure 3.10 shows the peak throughput of *Aliph*, PBFT and Zyzzyva

[7]Note that similar performance drops with large UDP packets have already been observed in the context of broadcast protocols. For instance, a recent study made by the authors of the JGroups toolkit showed that with 5K messages, their TCP stack achieves up to 5 times the throughput of their UDP stack, even if the latter includes some flow control mechanisms.

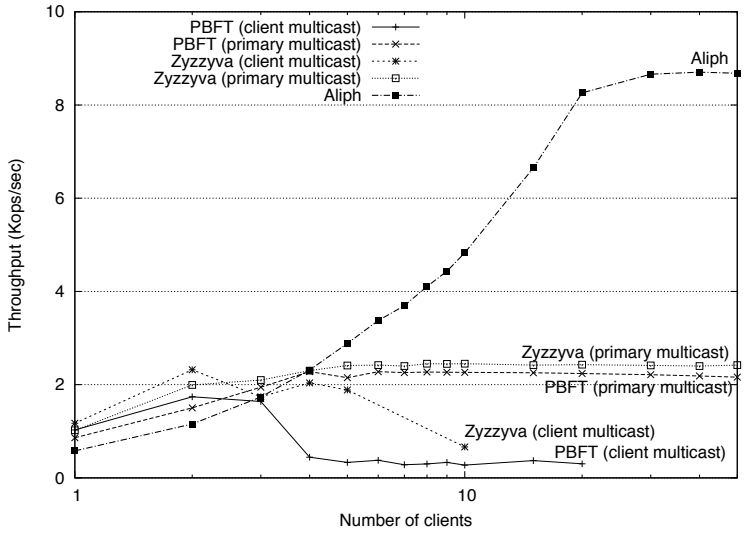


Figure 3.9: Throughput for the 4/0 benchmark (f=1).

as a function of the request size for $f = 1$. To obtain the peak throughput of PBFT and Zyzzyva, we benchmarked both protocols with and without client multicast optimization and with different batching sizes for Zyzzyva. Interestingly, the behavior we observe is similar to that observed using simulations in [87]: differences between PBFT and Zyzzyva diminish with the increase in payload. Indeed, starting from 128B payloads, both protocols have almost identical performance. Figure 3.10 also shows that *Aliph* sustains high peak throughput with all message sizes, which is again the consequence of *Chain* being active under contention.

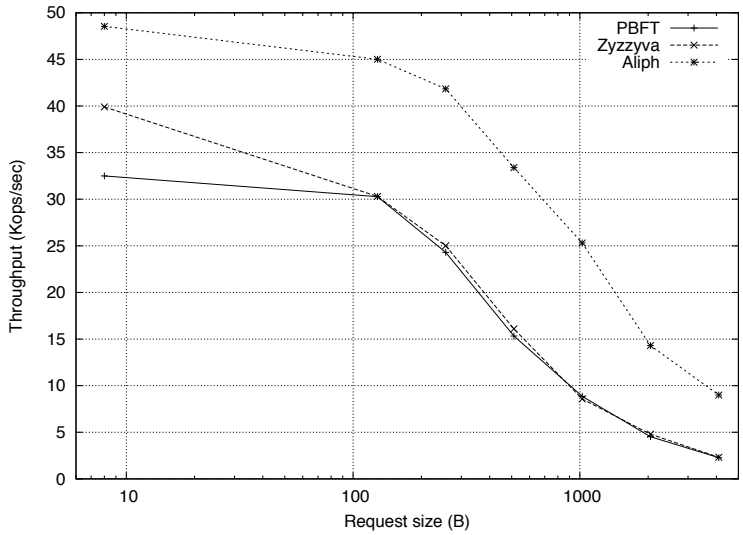


Figure 3.10: Peak throughput in function of request size (f=1).

Fault scalability. One important characteristic of BFT protocols is their behavior when the number of tolerated server failures f increases. Figure 3.11 depicts the performance of *Aliph* for the 4/0 benchmark when f varies between 1 and 3. We do not present results for PBFT

and Zyzzyva as their peak throughput is known to suffer only a slight impact [67]. Figure 3.11 shows that this is also the case for *Aliph*. The peak throughput at $f = 3$ is only 3,5% lower than that achieved at $f = 1$. We also observe that the number of clients that *Aliph* requires to reach its peak throughput increases with f. This can be explained by the fact that *Aliph* uses *Chain* under contention. The length of the pipeline used in *Chain* increases with f. Hence, more clients are needed to feed the *Chain* and reach the peak throughput.

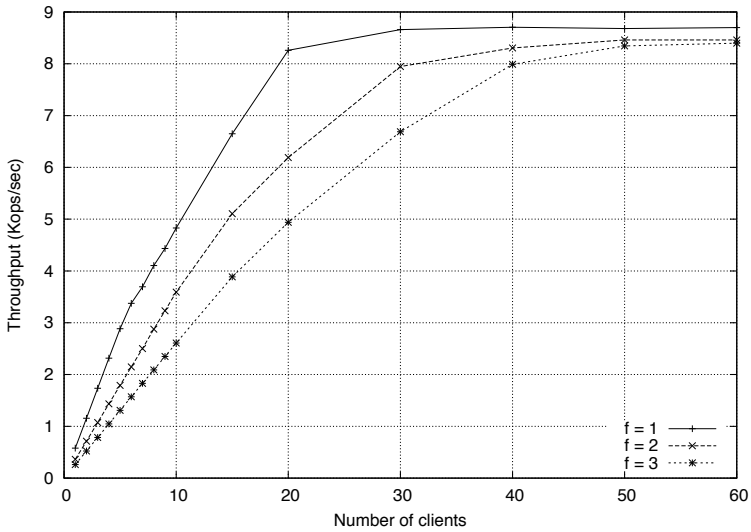


Figure 3.11: Impact of the number of tolerated failures f on the *Aliph* throughput.

3.5.2.3 Performance in Case of Faults

As mentioned in Section 3.4.3, when *Aliph* switches to *Backup*, it executes k requests, and then switches back to *Quorum*. To asses the impact of k on system's performance, we compared performance for two different values of k – in the first case, $k = 1$, and in the second case, $k = 2^i$, where i is the number of invocations of *Backup* since beginning.

In this experiment, we skip execution in *Chain*, and focus only on *Quorum* and *Backup*. There is one client, which issues 15.000 requests in total. If run just under *Quorum*, system would process these requests in 7s. After client sends 2.000 requests, one of the replicas goes down, and remains down for 10s. During this time, only three replicas are active. Hence, as *Quorum* protocol requires all replicas to respond, it will not execute any request. Figure 3.12 shows the throughput of the system, when *Aliph* switches to *Backup* for a single request. As discovery of replica failure in *Quorum* is done with timers, only handful of requests will be serviced (by *Backup*) while one replica is down. On the other hand, Figure 3.13 shows the behavior of the system if it stays in *Backup* for 2^i requests. Although it takes less time to finish the experiment, system may stay for too long in *Backup*. Replica came back up at $t = 11s$ in the experiment, but switch from *Backup* to *Quorum* occurred around $t = 14s$, because *Backup* had to process 8192 requests.

3.5.2.4 Dynamic Workload

Finally, we study the performance of *Aliph* under dynamic workload (i.e., fluctuating contention). We compare its performance to that achieved by Zyzzyva and by *Chain* alone. We do not present results for *Quorum* alone as it does not perform well under contention. The experiments consists in having 30 clients issuing requests of different sizes, namely, 0k,

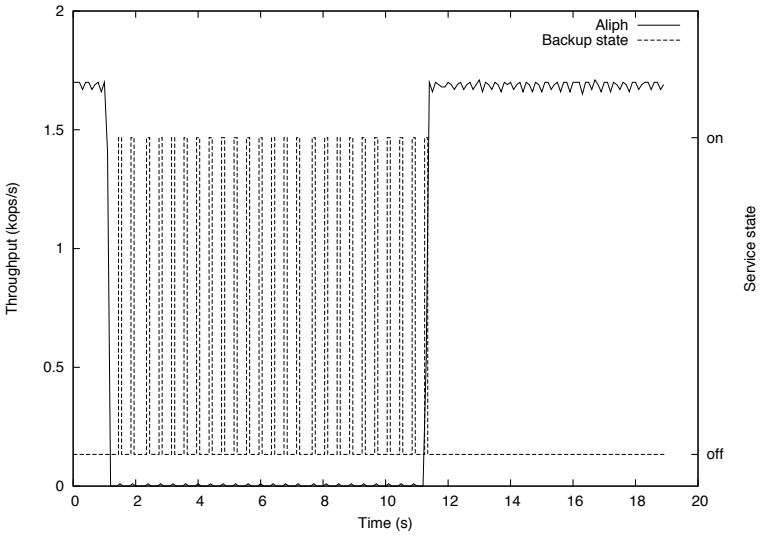


Figure 3.12: Throughput under faults, when system switches to *Backup* for one request.

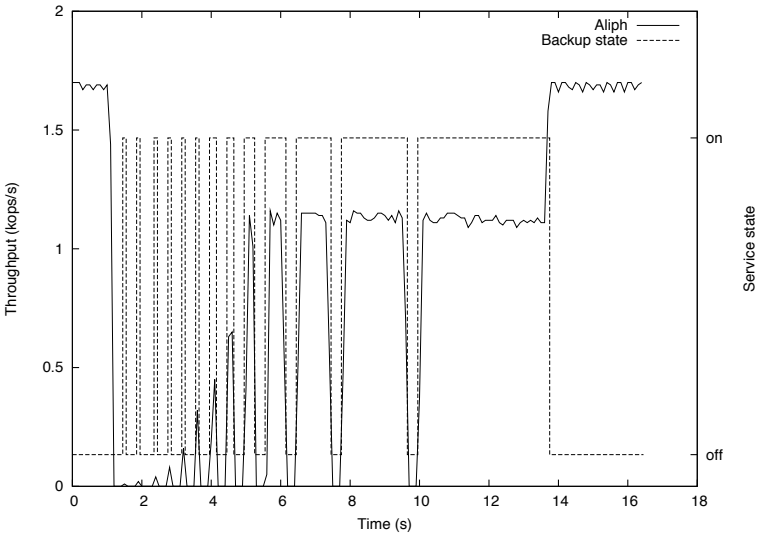


Figure 3.13: Throughput under faults, when system switches to *Backup* for 2^i requests.

0.5k, 1k, 2k, and 4k. Clients do not send requests all at the same time: the experiment starts with a single client issuing requests. Then we progressively increase the number of clients until it reaches 10. We then simulate a load spike with 30 clients simultaneously sending requests. Finally, the number of clients decreases, until there is only one client remaining in the system.

Figure 3.14 shows the performance of *Aliph*, Zyzzyva, and *Chain*. For each protocol, clients were invoking the same number of requests. Moreover, requests were invoked after the preceding clients had completed their bursts. First, we observe that *Aliph* is the most efficient protocol: it completes the experiment in $42s$, followed by Zyzzyva ($68.1s$), and *Chain* ($77.2s$). Up to time $t = 15.8s$, *Aliph* uses *Quorum*, which performs much better than Zyzzyva and *Chain*. Starting at $t = 15.8$, contention becomes too high for *Quorum*, which switches to *Chain*. At time $t = 31.8s$, there is only one client in the system. After $2s$ spent with only one client in the system, *Chain* in *Aliph* starts aborting requests due to the low load optimization (Sec. 3.5.1.3). Consequently, *Aliph* switches to *Backup* and then to *Quorum*. This explains the increase in throughput observed at time $t = 33.8s$. We also observe on the graph that *Chain* and *Aliph* are more efficient than Zyzzyva when there is a load spike: they achieve a peak throughput about three times higher than that of Zyzzyva. On the other hand, *Chain* and *Aliph* have slightly lower performance than Zyzzyva under medium load (i.e. from $16s$ to $26s$ on the *Aliph* curve). This suggests an interesting BFT protocol that would combine *Quorum*, Zyzzyva, *Chain* and *Backup*. However, this requires smart choices for dynamic switching, e.g., between Zyzzyva and *Chain*. We believe that building such a protocol is an interesting research topic.

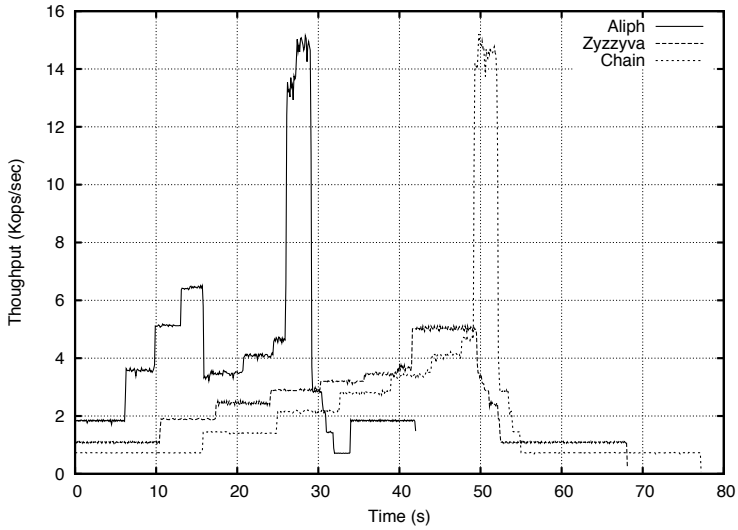


Figure 3.14: Throughput under dynamic workload.

3.6 Related Work

Several BFT protocols have been proposed in the last ten years with practical motivations in mind: *PBFT* [27] was the first to make a convincing argument for the practical potential of BFT. PBFT was in particular the first to use MAC vectors (instead of signatures) to improve performance. *Q/U* [1] was the first to implement the idea of a single communication round to expedite a decision in the absence of contention or timeouts. It required however $5f + 1$

servers to tolerate f Byzantine server failures, unlike our *Quorum* protocol which enables the same communication pattern with only $3f + 1$ servers.

Several examples of speculative protocols, distinguishing an optimistic phase from a recovery one, were discussed in the survey of Pedone [83]. These speculation ideas were used in the context of Byzantine state machine replication, e.g., in HQ [38] and Zyzzyva [67]. We are however the first to clearly separate the phases and encapsulate them within first class, well-specified, modules, that can each be designed, tested and proved independently. In a sense, Abstract enables to build a BFT protocol as the composition of as many (gracefully degrading) phases as desired, each with a "standard" interface. This allows for an unpreceded flexibility in BFT protocol design that we illustrated with *Aliph*, a BFT protocol that combines three different phases. In retrospect, and with *Abstract* in mind, one can obtain a modular protocol that mimics *Zyzzyva* (or *HQ*) in favorable cases for which it was optimized, by adding less than 24% to the code of a black-box BFT protocol (e.g. PBFT).

The idea of aborting if "something goes wrong" is almost as old as distributed computing. It underlies for instance the seminal two-phase commit protocol (2PC) [57]: abort can be decided if there is a failure or some database server votes negatively. Interestingly, our decomposition of a BFT protocol into several BFT sub-protocols with various progress semantics can be be viewed as way to obtain an effective non-blocking commit protocol [18], by combining several 2PC protocols (each optimized for a specific operating condition) together with a backbone 3PC protocol [88].

Two forms of *abortable agreement* were proposed in [32] and [22]. In the first case, a process can abort if a majority of processes cannot be reached whereas, in the second, a process can abort if there contention. The latter idea was generalized for arbitrary shared objects in [9] and then [2]. In particular, in [2], a process can abort and then query the object to seek whether the last query of the process was performed. This query can however abort if there is contention. Our abstraction is different in two senses. First, the condition under which *Abstract* can abort is a generic parameter: it can express for instance contention, synchrony or failures. Second, in case of abort, *Abstract* returns (without any further query) an information needed for recovery. This, in turn, can be used to invoke another, possibly stronger, *Abstract*. This ability is key to the composability of *Abstract*.

Several abstractions were proposed to deconstruct BFT protocols. In [45], a BFT protocol is viewed as a series of *weak-interactive consistency* instances. In [25], a BFT protocol is built using the abstraction of *multi-valued Byzantine agreement* with *external validity*. These abstractions are orthogonal to our and can, we believe, implement instances of *Abstract*. Neither of those tackle however the problem of composing several BFT protocols. The *optimistically terminating consensus (OTC)* framework proposed in [98] captures the notion of a round of communication and allows the deconstruction of single-instance agreement protocols. OTC seeks latency-optimal BFT protocols whereas *Abstract* is oblivious to any particular complexity metric. We indeed illustrated the benefits of *Abstract* with latency as well as throughput efficient BFT protocols.

Finally, let us note that there have been research works aiming at preserving what we call "common circumstances", even in the presence of attacks. Notably, some protocols [36, 34] have been devised by means of distributed systems models and architectures that limit, by construction, the attack model, that is to say, the potential of hackers to create deviations from "common circumstances". These works are complementary to the work presented in this chapter. One might devise Abstract instances following the principles described in these papers.

3.7 Conclusion

Progress in computer science comes with the design and implementation of abstractions that encapsulate specific aspects of a computation problem within a well-defined interface. Seminal examples include data structures like *records* and *arrays* for sequential programming, as well as synchronization primitives like *semaphores* and *monitors* for concurrent programming. Distributed programming abstractions do exist, such as *sockets* or *signatures*, but these are low-level and the difficulty of building and maintaining distributed systems calls for higher level abstractions. State machine replication might be viewed as one such high-level abstractions: it seeks to entirely hide distribution by providing the single, immortal, sever illusion. Yet, in some sense, once state machine replication is achieved, the job is over. What is lacking is a suite of abstractions to simplify the implementation and maintenance of state machine replication itself. We presented one such abstractions in this chapter [58, 95], with the aim to simplify the construction of the most resilient form of state machine replication protocols, namely BFT (Byzantine Fault-Tolerant) protocols.

The idea underlying our abstraction echoes Lampson's recommendation to build practically effective protocols by handling common and worst cases separately [71]. Clearly, what might be considered the common case for a given application and environment might not be considered common for another one. Even for the same application, what might be considered common might well change over time: the load that is considered normal for Amazon servers right before thanksgiving is not the same as the one that is considered normal during the early morning of January 5th. Our abstraction seeks to simplify the implementation of separate BFT sub-protocols, each intended to be executed under a specific common case. The key to simplicity is the observation that, precisely because many of those normal cases are favorable ones (e.g., no timeout, no failures or even sometimes no contention), there is no reason a BFT sub-protocol that needs to be executed only during those cases need to be as hard to implement, test and prove than a full-fledged BFT protocol, supposed to be performed under all conditions. The challenge to make that simplification possible was however to precisely define what needs to be ensured under a normal case and what exactly happens when the operating conditions are not those expected anymore: this is exactly what *Abstract* encapsulates. The genericity of *Abstract* captures the diversity of the very notion of a "normal case". Not surprisingly, certain *Abstract* instances are simple to implement and prove as they only need to guarantee progress under favorable circumstances. Yet, although simple to implement, some of those instances, such as our *Chain* sub-protocol, can drastically boost the peak throughput of a BFT protocol.

Needless to say, *Abstract* does not trivialize the problem of building effective BFT protocols. It does however give back to Caesar what belongs to Caesar. BFT experts can focus on building libraries of *Abstract* instances, whereas system experts can focus on analyzing which BFT protocol is best suited to a specific environment. Several directions can be interesting to explore. It would be interesting to devise efficient Abstract implementations for other interesting definitions of the progress property, e.g., implementations that perform well despite failures. It would be also interesting to explore possibilities for signature-free switching, to obtain practical BFT protocols that do not rely on signatures [37]. Moreover, we believe that an interesting research challenge is to define and evaluate effective heuristics for dynamic switching among Abstract instances. While we described *Aliph* and showed that, albeit simple, it outperforms existing BFT protocols, *Aliph* is simply the starting point for Abstract. The idea of dynamic switching depending on the system conditions seems very promising.

Chapter 4

Conclusion

This chapter concludes the thesis. We start by a brief summary of the contributions presented in this document. We then give some research directions for future work.

4.1 Summary of Contributions

This thesis makes two contributions to state machine replication, a software technique aiming at building fault-tolerant systems using commodity hardware. The operating principle of state machine replication is very simple: the service to be made fault-tolerant is replicated on different machines. A state machine replication protocol ensures that all replicas receive the same set of requests in the same order.

The first contribution is LCR [60, 61, 73], a uniform total order broadcast protocol that can be used to implement a state machine replication protocol in an environment where nodes only fail by crashing, and where crashes can be accurately detected (using a failure detector). When executed on a set of homogeneous machines interconnected by a LAN, LCR ensures optimal throughput during failure-free periods. To the best of our knowledge, LCR is the first total order broadcast protocol to achieve optimal throughput in this environment. Contrarily to most existing protocols, LCR does not rely on a native multicast primitive, but rather uses point-to-point communication channels. More precisely, LCR organizes processes in a ring topology: each node has only one successor in the ring to which it forwards messages. We show both theoretically and experimentally that this ring topology yields optimal throughput. Moreover, we benchmark an implementation of LCR and we show that it outperforms the two most widely used total order broadcast protocols.

The second contribution presented in this thesis is Abstract [58, 95], a novel abstraction that simplifies the design, implementation, testing and verification of state machine replication protocols. In particular, Abstract focuses on state machine replication protocols tolerating arbitrary (also called Byzantine) faults. In a sense, Abstract is an abortable state machine that enables to build a state machine replication protocol as the composition of as many (gracefully degrading) phases as desired, each with a "standard" interface. These phases are Abstract instances and each of them can be designed, implemented, tested and proved independently. This allows for an unpreceeded flexibility in state machine replication protocol design that we illustrated with the development of two protocols. The first one mimics a state of the art protocol, but requires writing much less code (only 24% of the legacy code size). This is due to the fact that Abstract's modularity allows easily reusing and factorizing code. The second protocol we developed using Abstract is called *Aliph*. It illustrates the ability to develop very efficient protocols using this modular approach. In particular, we evaluated *Aliph* using microbenchmarks and showed that it outperforms state of the art protocols both

in terms of latency (by up to 30%) and throughput (by up to 360%).

4.2 Future Work

State machine replication has been a very active research topic for more than 20 years. Yet, we believe that many works remain to be done. In the remainder of this section we discuss three of them.

Wide-area Byzantine-resilient state machine replication. Services are increasingly migrating to data centers geographically spread across the Internet. This is the so-called "cloud" architecture. The benefits of cloud architectures are numerous. Let us mention, for instance, the ability for users to access services from any host or the possibility to automate both service upgrades and data backups.

Unfortunately, a whole data center can be rendered unavailable by a catastrophic event, a loss of power, or simply a loss of connectivity. For reliability reasons, it is thus not a good idea to execute a service on a single data center. Moreover, it is often desired to decrease the latency perceived by clients. This can be achieved by locating a service on several data centers geographically spread around the world in order to decrease the probability to encounter network congestion between clients and the service.

It is thus becoming necessary to devise state machine replication protocols for services hosted on multiple clouds. We believe that it is important that these protocols tolerate Byzantine faults as it does not seem reasonable to assume only crash failures in large-scale environments, where nodes are hosted on different clouds and administered by different entities. BFT protocols studied in Chapter 3 are not good candidates for such environments. Indeed, they do either not tolerate contention [38, 1], or they rely on a leader to allocate sequence numbers [67, 27]. Requests generated by clients close to the leader might be favored by such protocols. Moreover, leader-based protocols have unbalanced communication patterns that limit the utilization of the available bandwidth. Finally, these protocols rely on IP multicast, which is known to perform poorly in large-scale settings.

To the best of our knowledge, only two BFT protocols have been specifically devised for wide-area networks. The first one, called Steward [6], is a hierarchical BFT protocol that combines a BFT protocol running on each site (or cloud) and a Paxos-like protocol between the different sites. The main drawback of this protocol is that it requires a minimum of 3f+1 machines running in each cloud. Another weakness is that the Paxos-like protocol running between the different sites is not optimized for wide-area networks. The second protocol is proposed by Amir *et al.* [5]. The paper describes a mechanism that allows establishing a reliable virtual communication link between sets of machines located in different sites. The goal of this mechanism is to have all machines of a given site appear as being a single machine. A traditional replication protocol can then be used between the different sites. This replication protocol needs to deal with a reduced number of nodes (only one virtual node per site). Like Steward, the protocol used for inter-site replication is not optimized for wide-area networks.

We would like to propose BFT protocols optimized for wide-area communications. Such protocols must provide both high throughput under high client load and low latency under low client load. They must also be able to face changing network conditions and varying client load. Finally, they must be able to leverage asymmetric network link performance. A path we would like to explore is the use of multiple leaders to assign distinct sets of sequence numbers. This idea has been initially proposed by Mao *et al.* [78] to tolerate benign failures. We believe that it could be adapted to tolerate Byzantine failures. Another path we envision is the combination of multiple communication patterns (e.g. tree, multicast, pipe-line) in order to deal with the heterogeneity of environments comprising multiple clouds. For instance, one

might combine the pipelining pattern used in the Chain protocol presented in Chapter 3 with the multicast-based pattern used in Zyzzyva [67].

Parallel state machine replication. Hardware technology trends indicate that processor clock rates are stalling and that future performance improvements are likely to come from increased parallelism rather than raw speed improvements. As hardware parallelism increases, services are expected to become more and more parallel in order to provide better performance.

Unfortunately, state machine replication systems have traditionally required deterministic execution achieved through sequential state machine execution. This means that current techniques for state machine replication are only applicable to single-threaded applications and cannot safely leverage hardware parallelism. This is a real and major performance penalty considering that it is now admitted that the performance bottleneck in replicated state machines is the execution of the state machine rather than the coordination of replicas.

There have been proposals to avoid the limitations of sequential state machine execution. For instance, it has been suggested to weaken reliability guarantees in order to allow replica states to temporarily diverge [79, 90]. This approach can nevertheless not be applied to services, such as databases, that require strong consistency. Another approach is to enforce determinism at the operating system level. Recently, an operating system called Determinator [10], has been proposed. Determinator enforces determinism on both multi-threaded and multi-process computations. It can run parallel applications deterministically both on multi-core PCs and across nodes in a cluster. This approach works well for applications exhibiting coarse-grained parallelism, but it induces a high overhead for fine-grained parallel applications. Finally, a third approach has been described in [68], where authors propose to use application-specific information to identify and concurrently execute independent requests. They design an architecture in three stages responsible for agreement on request order, request parallelization, and request execution, respectively. The parallelization stage uses rules supplied by the application designer to identify the set of requests that can be executed in parallel without compromising safety. This approach has a major drawback: it requires rules to be correct, otherwise, safety is not ensured. Writing correct rules might not be trivial. Consider for instance a database that implements row-level locking. In order to achieve the highest level of parallelism, rules should be based on the set of rows that each database query will update, which can be extremely hard to know at design time.

We believe that state machine determinism is not necessarily exclusive with parallel execution. We plan to study the possibility to design an architecture that let replicas optimistically execute requests and then verify whether or not they have produced equivalent results. This is in contrast with current state machine replication architectures that, even when speculative [67], first agree on the order in which requests will be executed and then rely on deterministic execution to guarantee that all replicas reach the same state. Embracing the mantra of "trust but verify" should allow leveraging benign parallelism, but it will introduce additional cost and complexity for higher degrees of parallelism. Specifically, two replicas that execute requests in parallel may produce different outputs based on differences in local thread scheduling. When replicas diverge, it will be necessary to invoke repair mechanisms in order to maintain consistency. In order for repair to be efficient, we will need to implement various mechanisms such as incremental state rollback, fast checkpoint generation and comparison, and fast incremental state transfer.

Single-node Byzantine-resilient state machine replication. The number of standard processing cores per chip doubles with every semi-conductor process generation. Unfortunately, the increase in computing power has not been followed by an increase in software and hardware reliability. Zhenmin Li et al. have recently presented a study of around 29,000 bugs in Mozilla and Apache Web Server [74]. They observe that around 10% of Apache bugs lead to

a crash and that bugs related to the security of the application are increasingly present over the years. On the hardware side, core failures such as manufacturing defects and soft errors are becoming a concern. This comes from the fact that the number of transistors put into a processor drastically increases, but not the reliability of each individual transistor.

We have seen in this thesis that one way to improve the reliability of software systems is to use replication. We have studied various protocols that allow running several (possibly different) copies of a state machine on different machines. We believe that an interesting research topic is to study the possibility to implement state machine replication on a single-machine. The idea is to collocate the different replicas of the state machine on the same physical machine in order to take advantage of the increasing power of modern computers to improve the resilience of software systems. Note that the idea of considering a single computer as a distributed system is a current research trend. For instance, Baumann *et al.* argue that modern computers resemble a distributed system and propose Barrelfish [17], an operating system designed as a distributed system. Note also that implementing state machine replication on a single machine has been studied by Chun *et al.* in a position paper [33]. This paper does nevertheless not propose a solution. It only describes challenges that need to be solved and a few potential designs.

Implementing state machine replication on a single machine will open new perspectives for the design of replication protocols and underlying communication mechanisms. Indeed, if current BFT replication protocols can easily run on a multi-core machine, they will most probably not achieve the best possible performance. The reasons are twofold. First, these protocols have been designed for asynchronous systems. On a multicore machine, it is maybe possible to make some synchrony assumption on processing time and communications. More precisely, if we assume that memory and I/O buses are fault-free and that replicas are equally fast, it is possible to assume a synchronous system model. Second, current BFT protocols rely on existing communication mechanisms, such as Inet sockets, that have been designed for communications between remote machines. These mechanisms can be used by processes running on the same machine. Nevertheless, their performance will not be as good as what could be achieved by customized mechanisms. We believe that it will thus be necessary to design a new communication mechanism implementing a very efficient and robust multicast primitive for single-machine communications. This mechanism cannot be implemented using shared memory techniques (as done in Barrelfish [17]) since it needs to be robust, i.e. to ensure that messages cannot be forged by erroneous replicas.

Bibliography

[1] Michael Abd-El-Malek, Gregory Ganger, Garth Goodson, Michael Reiter, and Jay Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[2] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical report, CNDS-2004-1, Johns Hopkins University, 2004.

[4] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[5] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Customizable Fault Tolerance for Wide-Area Replication. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 65–82, October 2007.

[6] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.

[7] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[8] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. RFC 1301, IETF, 1992.

[9] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC*, 2005.

[10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the International Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] R. Baldoni, S. Cimmino, and C. Marchetti. A Classification of Total Order Specifications and its Application to Fixed Sequencer-based Implementations. *to appear in Journal of Parallel and Distributed Computing*, 6 2006.

[12] Roberto Baldoni, Roberto Beraldi, Vivien Quéma, Leonardo Querzoni, and Sara Tucci-Piergiovanni. TERA: topic-based event routing for peer-to-peer architectures. In *Proceedings of the International Conference on Distributed Event-based Systems (DEBS)*, pages 2–13, New York, NY, USA, 2007. ACM.

[13] Roberto Baldoni, Rachid Guerraoui, Ron Levy, Vivien Quéma, and Sara Tucci Piergiovanni. Unconscious Eventual Consistency with Gossips. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 4280 of *Lecture Notes in Computer Science*, pages 65–81, Dallas, TX, USA, November 2006. Springer.

[14] Bela Ban. *JGroups – A Toolkit for Reliable Multicast Communication*. http://www.jgroups.org, 2007.

[15] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[16] Amotz Bar-Noy and Shlomo Kipnis. Multiple message broadcasting in the postal model. *Networks*, 29(1):1–10, 1997.

[17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, pages 29–44, New York, NY, USA, 2009. ACM.

[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87)*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[20] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[21] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[22] Romain Boichat, Partha Dutta, Svend Frölund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News in Distributed Computing*, 34(1):47–67, 2003.

[23] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *PaCT*, 2001.

[24] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12):1257–1284, September 2006.

[25] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *CRYPTO*. Springer, 2001.

[26] R. Carr. The tandem global update protocol. *Tandem Syst. Rev. 1*, pages 74–85, jun 1985.

[27] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[28] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[29] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[30] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[31] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[32] Wei Chen. Abortable consensus and its application to probabilistic atomic broadcast. Technical Report MSR-TR-2006-135, 2007.

[33] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse Replication for Single-Machine Byzantine-Fault Tolerance. In *Proceedings of the USENIX Annual Technical Conference*, pages 287–292, June 2008.

[34] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, pages 189–204, October 2007.

[35] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[36] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, October 2004.

[37] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.*, 49(1):82–96, 2006.

[38] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[39] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[40] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distrib. Syst. Eng. J.*, 4(2):109–128, jun 1997.

[41] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[42] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[43] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[44] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *DSN*, 2006.

[45] Assia Doudou. *Abstractions for Byzantine-resilient state machine replication*. PhD thesis, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland, February 2000.

[46] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. Fuse: Lightweight guaranteed distributed failure notification. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[47] R. Ekwall, A. Schiper, and P. Urban. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 52–65, Washington, DC, USA, 2004. IEEE Computer Society.

[48] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, Washington, DC, USA, 1995. IEEE Computer Society.

[49] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous gossip. In *Proceedings of the International Conference on Middleware (Middleware)*, pages 42–61, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[50] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod, and Vivien Quéma. Stretching gossip with live streaming. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2009.

[51] T. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[52] Toy Friedman and Robbert Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 233, Washington, DC, USA, 1997. IEEE Computer Society.

[53] U. Fritzke, P. Ingels, A. Mostefaoui, and M. Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):147–156, 2001.

[54] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst.*, 9(3):242–271, 1991.

[55] Fabien Gaud, Sylvain Geneves, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Efficient Workstealing for Multicore Event-Driven Systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 516–525. IEEE Computer Society, June 2010.

[56] A. Gopal and S. Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 110–123, London, UK, 1989. Springer-Verlag.

[57] Jim Gray. Notes on database operating systems. In *Operating Systems — An Advanced Course*, number 66. 1978.

[58] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, pages 363–376, New York, NY, USA, 2010. ACM.

[59] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quéma. A High Throughput Atomic Storage Algorithm. In *The 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, 2007.

[60] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. High Throughput Total Order Broadcast for Cluster Environments. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, Philadelphia, PA, USA, 2006.

[61] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):1–32, 2010.

[62] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. pages 97–145, 1993.

[63] Rick Jones. *Netperf*. http://www.netperf.org/, 2007.

[64] F. Kaashoek and A. Tanenbaum. An evaluation of the amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[65] Anne-Marie Kermarrec, Alessio Pace, Vivien Quéma, and Valerio Schiavoni. NAT-resilient Gossip Peer Sampling. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, June 2009.

[66] J. Kim and C. Kim. A total ordering protocol using a dynamic token-passing scheme. *Distrib. Syst. Eng. J.*, 4(2):87–95, jun 1997.

[67] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, 2007.

[68] Ramakrishna Kotla and Michael Dahlin. High Throughput Byzantine Fault Tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584, June 2004.

[69] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[70] Leslie Lamport. Lower bounds for asynchronous consensus. In *FuDiCo*, 2003.

[71] Butler W. Lampson. Hints for computer system design. In *SOSP*, pages 33–48, 1983.

[72] Matthieu Lerclercq, Ali Erdem Özcan, Vivien Quéma, and Jean-Bernard Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2007.

[73] Ron Levy. *The complexity of reliable distributed storage*. PhD thesis, EPFL, Lausanne, 2008.

[74] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[75] S. Luan and V. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):271–285, 1990.

[76] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[77] L. Malhis, W. Sanders, and R. Schlichting. Numerical performability evaluation of a group multicast protocol. *Distrib. Syst. Enj. J.*, 3(1):39–52, march 1996.

[78] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Proceedings of the International Symposium on Operating Systems Design and Implementation (OSDI)*, pages 369–384, 2008.

[79] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of International Symposium on Principles of Distributed Computing (PODC)*, pages 108–117, 2002.

[80] Sonia Ben Mokhtar, Alessio Pace, and Vivien Quéma. FireSpam: Spam Resilient Gossiping in the BAR Model. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, October 2010.

[81] L. Moser, P. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput.*, 22(4):727–750, 1993.

[82] T. Ng. Ordered broadcasts for large applications. In *Proceedings of the 10th IEEE International Symposium on Reliable Distributed Systems (SRDS'91)*, pages 188–197, Pisa, Italy, 1991. IEEE Computer Society.

[83] Fernando Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[84] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, 1989.

[85] L. Rodrigues, H. Fonseca, and P. Verï¿½ssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[86] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[87] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In *NSDI*, 2008.

[88] D. Skeen. Nonblocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM Press, 1981.

[89] J. R. Stanton. *A Users Guide to Spread*. http://www.spread.org/docs/guide/users_guide.pdf, 2002.

[90] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, pages 172–182, New York, NY, USA, 1995.

[91] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, pages 582–589, 2000.

[92] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[93] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[94] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[95] Marko Vukolić. *Abstractions for asynchronous distributed computing with malicious players*. PhD thesis, EPFL, Lausanne, 2008.

[96] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 33–57, London, UK, 1994. Springer-Verlag.

[97] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1995. IEEE Computer Society.

[98] Piotr Zielinski. *Minimizing latency of agreement protocols*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK, September 2005.

Appendix A

Abstract: specification

Before diving into the precise specification of Abstract, we first introduce some notations and definitions. We denote the output function of the (replicated) state machine by $rep(h)$, where h is a sequence of requests called *commit history*: the initial state is an implicit argument of $rep()$. Basically, $rep(h)$ represents *replies* that state machine outputs to clients. We assume that every Abstract instance has its own unique id i (a natural number) that uniquely determines the implementation and the set of underlying servers.

Abstract i exports one operation: $Invoke_i(m, [h])$, where m is a request, and h a (optional) sequence of requests called *init history*; we say the client *invokes* request m (with init history h). By convention, when $i = 1$, the invocation never contains an init history. Abstract i returns two indications to the client:

1. $Commit_i(m, rep(h))$, where commit history h contains m;

2. $Abort_i(m, h, next(i))$, where the sequence of requests h is called an *abort history* and $next$ is a function that returns an integer.

Respectively, we say that a client *commits/aborts* the request m. In the case of an abort, we also say that instance Abstract i *switches* to instance $next(i)$. We also define a *valid init history* (VIH) as follows: init history h is a VIH if there is an Abstract j that returned $Abort_j(*, h, i)$ (i.e., such that $next(j) = i$). Similarly, we define a *valid init request* (VIR) as follows: (1) if $i = 1$, any invoked request is a VIR; (2) if $i \neq 1$, m is a VIR if and only if m is invoked with a VIH. We say that an instance $i > 1$ is *initialized* upon it commits or aborts some VIR. Finally, we say that a request m is *valid* if (1) m is a VIR, or (2) m is invoked after i gets initialized.

We are now ready to specify the properties of Abstract (parametrized by a predicate P). In the following, "prefix" refers to a non-strict prefix.

1. *(Validity)* In every commit/abort history, no request appears twice and every request is a valid request, or an element of a valid init history.

2. *(Termination)* If a correct client c invokes a valid request m, then c eventually commits or aborts m.

3. *(Progress)* If a correct client c invokes a valid request m and some predicate P holds, then c commits m.

4. *(Init Order)* The longest common prefix of all valid init histories is a prefix of any commit or abort history.

5. *(Commit Order)* Let h and h' be any two commit histories: either h is a prefix of h' or vice versa.

6. *(Abort Order)* Every commit history is a prefix of every abort history.

7. *(Switching Monotonicity)* For every Abstract instance i, $i < next(i)$.

It is important to see that Abstract is a strict generalization of BFT. Namely, BFT is precisely an Abstract (with id $i = 1$) that never aborts. In this case, *Abort Order* and *Switching Monotonicity* become irrelevant, and *Init Order* trivially holds.

We build BFT protocols by composing Abstract instances. At the heart of the composition lies a simple scheme (we refer to as Abstract composition algorithm (ACA)), given in Figure A.1, where: a) correct clients use an abort history of an aborting Abstract instance (e.g., i) as the init history for the next instance ($next(i)$), and b) a given client invokes $next(i)$ with an init history included only once (with its first invocation of $next(i)$): subsequently, it invokes $next(i)$ without any init history.

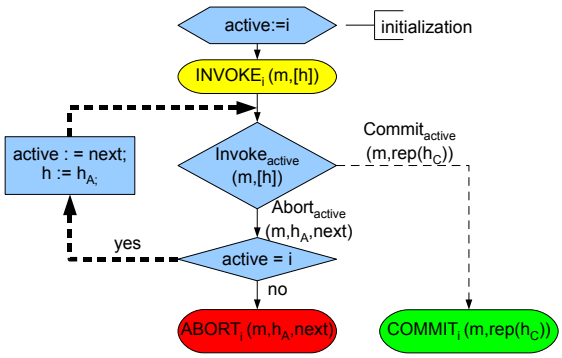


Figure A.1: Abstract composition algorithm (ACA). Invocations/indications of the composed Abstract are in uppercase.

The key invariant in the Abstract framework is *idempotence*: a composition of *any two* Abstract instances is, itself, an Abstract instance. More precisely:

Theorem 1 *Given any two Abstract instances i and i' and integer i'', such that $i' = next(i)$ and $next(i') = i''$, ACA implements a single* Abstract *instance with the instance id i, such that $next(i) = i''$.*

By induction, the theorem extends to a composition of *any* number of Abstract instances, which yields again an Abstract instance.

Appendix B

Abstract: detailed protocol descriptions

In this Appendix, we give the details of *ZLight*, *Quorum*, *Chain* and their shared panicking mechanism (Sec. B.2 - B.5). For ease of presentation, we first give details about handling invocations without init histories and then (Sec. B.6) we show how we handle init histories and switch between Abstract instances in *AZyzzyva* and *Aliph*. Finally, we give the details of our checkpointing subprotocol in Section B.6.1. Correctness proofs can be found in Appendix C.

B.1 Notations

A message m sent by process p to the process q and authenticated with a MAC is denoted by $\langle m \rangle_{\mu_{p,q}}$. A process p can use vectors of MACs (called authenticators [27]) to simultaneously authenticate the message m for multiple recipients belonging to the set S; we denote such a message, which contains $\langle m \rangle_{\mu_{p,q}}$, for every $q \in S$, by $\langle m \rangle_{\alpha_{p,S}}$. In addition, we denote the digest of the message m by $D(m)$, whereas $\langle m \rangle_{\sigma_p}$ denotes a message that contains $D(m)$ signed by the private key of process p and the message m. All processes are assumed to own the public key of every other process. Finally, we denote the set of all $(3f+1)$ replicas by Σ.

i - current Abstract id
c/r_j - client (resp., replica) ID
t_c - local timestamp at client c
$t_j[c]$ - the highest t_c seen by replica r_j
o - operation invoked by the client
LH_j - a local history at replica r_j
$reply_j$ - $rep(LH_j)$ (application reply in function of LH_j)
sn_j - sequence number at replica r_j (not used in *Quorum*)

Figure B.1: Message fields and local variables.

Notation for message fields and client/replica local variables used in *ZLight*, *Quorum* and *Chain* is shown in Figure B.1. To help distinguish clients' requests for the same operation o, we assume that client c calls $Invoke_i(req)$, where $req = \langle o, t_c, c \rangle$ and where t_c is a unique, monotonically increasing client's timestamp. A replica r_j *executes* req by appending it to LH_j.

B.2 ZLight Details

Step Z1. On $Invoke_i(req)$, client c sends the message $m' = \langle \mathsf{REQ}, req, i \rangle_{\alpha_{c,\Sigma}}$ to the primary (say r_1) and triggers timer T.

Step Z2. The primary r_1 on receiving $m' = \langle \mathsf{REQ}, req, i \rangle_{\alpha_{c,\Sigma}}$, if:

- $req.t_c$ is higher than $t_1[c]$,

then it:

- updates $t_1[c]$ to $req.t_c$,
- increments sn_1, and
- sends $\langle \langle \mathsf{ORDER}, req, i, sn \rangle_{\mu_{r_1,r_j}}, MAC_j \rangle$ to every replica r_j, where MAC_j is the MAC entry for r_j in the client's authenticator for m'.

Step Z3. Replica r_j on receiving (from primary r_1)
$\langle \langle \mathsf{ORDER}, req, i, sn' \rangle_{\mu_{r_1,r_j}}, MAC_j \rangle$, if:

- MAC_j authenticates req and i,
- $sn' = sn_j + 1$, and
- $t_j[c] < req.t_c$,

then it:

- updates sn_j to sn' and $t_j[req.c]$ to $req.t_c$,
- executes req, and
- sends $\langle \mathsf{RESP}, reply_j, D(LH_j), i, req.t_c, r_j \rangle_{\mu_{r_j,c}}$ to c[1].

Moreover, if MAC_j verification fails, r_j stops executing Step Z3 in instance i.

Step Z4. If client c receives $3f + 1$ $\langle \mathsf{RESP}, reply, LHDigest, i, req.t_c, * \rangle_{\mu_{*,c}}$ messages from different replicas before expiration of T, with identical digests of replicas' local history ($LHDigest$) and identical replies (or digests thereof), then the client commits req with $reply$. Otherwise, the client triggers the panicking mechanism explained in Section B.5 (Step P1).

B.3 Quorum Details

Step Q1. On $Invoke_i(req)$, client c sends message $\langle \mathsf{REQ}, req, i \rangle_{\mu_{c,\Sigma}}$ to all replicas and triggers timer T.

Step Q2. Replica r_j on receiving $\langle \mathsf{REQ}, req, i \rangle_{\mu_{c,\Sigma}}$ from client c, if:

- $req.t_c$ is higher than $t_j[c]$

then it:

- updates $t_j[c]$ to $req.t_c$,
- executes req, and
- sends $\langle \mathsf{RESP}, reply_j, D(LH_j), i, req.t_c, r_j \rangle_{\mu_{r_j,c}}$ to c.

Step Q3. Identical to Step Z4 of *ZLight*.

[1] As an optimization (which also applies to Step Q2 of *Quorum*), all but one designated replica can send reply digests $D(reply_j)$ instead of $reply_j$.

B.4 Chain Details

In the following, we assume that every CHAIN message sent by process p contains its respective chain authenticator (CA), as well as MACs p received from its predecessor $\overleftarrow{p}$ destined to processes in p's successor set (see Sec. 3.5.1.2).

Step C1. On $Invoke_i(req)$, client c sends the message $m' = \langle \mathsf{CHAIN}, req, i \rangle$ to the head (say r_1) and triggers the timer T.

Step C2. The head r_1, on receiving $m = \langle \mathsf{CHAIN}, req, i \rangle$ from client c, if:

- $req.t_c$ is higher than $t_1[c]$, and
- the head can verify client's MAC (otherwise the head discards m),

then the head:

- updates $t_1[c]$ to $req.t_c$,
- increments sn_1, and
- sends $\langle \mathsf{CHAIN}, req, i, sn_1, \perp \rangle$ to $\overrightarrow{r_1} = r_2$.

Step C3. Replica r_j on receiving $m = \langle \mathsf{CHAIN}, req, i, sn, REPLY \rangle$ from $\overleftarrow{r_j}$, if

- it can verify MACs from all processes from its predecessor set against the content of m,
- $sn = sn_j + 1$, and
- $req.t_c$ is higher than $t_j[c]$,

then it:

- updates sn_j to sn and $t_j[c]$ to $req.t_c$,
- (ii) executes req, and
- (iii) sends $\langle \mathsf{CHAIN}, req, i, sn, REPLY, LHDigest \rangle$ to $\overrightarrow{r_j}$, where $REPLY = LHDigest = \perp$ in case of the first $2f$ replicas, $REPLY = D(reply_j)$ and $LHDigest = D(LH_j)$ in case $i \in \{2f+1 \ldots 3f\}$, or $REPLY = reply_j$ and $LHDigest = D(LH_j)$ in case r_j is tail.

In case MAC verification mentioned above fails, replica stops executing Step C3 in instance i.

Step C4. If client c receives $\langle \mathsf{CHAIN}, req, i, *, reply, LHDigest \rangle$ from the tail before expiration of T, and with MACs from last $f+1$ replicas that authenticate req, i, $LHDigest$ and $D(reply)$ (or $reply$ itself), then c commits req with $reply$. Otherwise, the client triggers the panicking mechanism explained in the following section (Step P1, Sec. B.5).

B.5 Panicking Mechanism

This is the mechanism through which we initiate the switching from *ZLight* (resp., *Quorum*, *Chain*) in Step Z4 (resp., Q3, C4).

Step P1. If the client does not commit request req by the expiration of timer T (triggered in Steps Z1/Q1/C1), c panics, i.e., it sends a $\langle \mathsf{PANIC}, req.t_c \rangle_{\mu_{c,r_j}}$ message to every replica r_j. Since messages may be lost, client periodically PANIC messages , until it aborts the request.

Step P2. Replica r_j, on receiving a $\langle \mathsf{PANIC}, req.t_c \rangle_{\mu_{c,r_j}}$ message, stops executing new requests (i.e., stops executing Step Z3/Q2/C3) and sends $\langle \mathsf{ABORT}, req.t_c, LH_j, next(i) \rangle_{\sigma_{r_j}}$ to c.

Step P3. When client c receives $2f+1$ $\langle \mathsf{ABORT}, req.t_c, *, next(i) \rangle$ messages with correct signatures from different replicas and the same value for $next(i)$, the client collects these messages into the set $Proof_{AH_i}$, and extracts the abort history AH_i from $Proof_{AH_i}$ as follows:

- First, c generates history h such that $AH[j]$ equals the value that appears at position $j \geq 1$ of $f+1$ different histories LH_j that appear in $Proof_{AH_i}$;
- If such a value does not exist for position x then h does not contain a value at position x or higher.
- Finally, AH_i is the longest prefix of h in which no request appears twice.

B.6 Handling Init Histories and Switching

To switch from *ZLight, Quorum* or *Chain* instance i, client invokes instance $i' = next(i)$ by accompanying req with init history $IH_{i'} = AH_i$ and $Proof_{AH_i}$. Then a replica running instance i', executing its first request in i' (e.g., in Step C3 of *Chain*), simply makes the library call to verify IH'_i against $Proof_{AH_i}$ following the algorithm given in Sec. B.5 and verifies that ABORT messages in $Proof_{AH_i}$ indeed declare i' as $next(i)$. In the case of switching to *Backup*, this check is simply a part of the functionality implemented on top of the underlying BFT.

To switch from *Backup*[2], *Backup* replicas must provide the client with $f+1$ different signatures of the identical abort history and the next Abstract instance id i'. This is a reasonable requirement on the BFT that underlies *Backup*, since any BFT protocol must anyway provide an identical reply from at least $f+1$ replica; in the case of *Backup* abort history, we just require this particular reply to be signed (we trivially implemented this in PBFT). Then, the client includes these signatures with req in its invocations of an Abstract i' (e.g., *ZLight*) and replicas running i', before executing the request (e.g., in Step Z3 of *ZLight*), simply verify the signatures against the submitted init history to find $f+1$ matching ones.

In all cases of switching, a replica running instance i' executes all the requests contained in the *first verified* init history IH'_i before executing the invoked request itself. Replicas simply ignore all subsequent init histories. Below we summarize the additional init history related actions performed by processes in steps of *ZLight*, *Quorum* and *Chain*. In the following, we assume that the verification of init histories is performed as described above.

[2] In *AZyzzyva*, *Backup* switches to *ZLight*, whereas in *Aliph* it switches to *Quorum*.

Step Z1/Q1/C1. On $Invoke_{i'}(req, IH)$, the message(s) sent by the client contain also IH and the set of signatures $Proof_{IH}$ returned by the preceding Abstract i, where $i' = next(i)$.

Step Z2/C2. If its local history LH_1 is empty, the primary/head r_1 executes the step only if IH can be verified against $Proof_{IH}$.

Step Z3/Q2/C3. If its local history LH_j is empty, the replica r_j executes the step only if it receives IH that can be verified against $Proof_{IH}$. If so, then (before executing req) r_j executes all the requests contained in IH (i.e., r_j sets LH_j to IH); then r_j executes req unless req was already in IH.

Step P1. On sending PANIC messages for a request that was invoked with an init history, client also includes IH and the set of signatures $Proof_{IH}$ returned by the preceding Abstract i within a PANIC message.

Step P2. If its local history LH_j is empty, replica r_j, executes the step only if IH can be verified against $Proof_{IH}$. Then, before executing the step as described in Section B.5, r_j first sets LH_j to IH.

B.6.1 Lightweight Checkpointing Subprotocol

In *ZLight*, *Quorum* and *Chain* we use a *lightweight checkpoint subprotocol* (LCS) to truncate histories every CHK requests (in our evaluations, $CHK = 128$), similarly to checkpoint protocols used in [27, 67]. Here, we explain our simple LCS and its impact on our implementations as presented earlier in this appendix.

LCS consists in the following:

1. every replica r_j increments the checkpoint counter cc and sends it along with the digest of its local state to every other replica (using simple point-to-point MACs), when its (non-checkpointed suffix of) local history reaches CHK requests. Then, r_j triggers a checkpoint timer.

2. if the timer expires and there is no checkpoint, the replica stops executing all requests.

3. If replica r_j receives the digest of the same state st with the same checkpoint counter number cc greater than $lastcc$ (initially $lastcc = 0$) from *all* replicas, r_j: (a) truncates its local history and checkpoints its state to st and (b) stores cc to variable $lastcc$. Such a checkpointed state (referred to as st_{cc}) becomes a prefix of replicas' local histories to which new requests are appended and is treated as such in all operations on local histories in our algorithms. Moreover, every abort or commit history of length at most $cc * CHK$ is considered to be a prefix of st_{cc}.

LCS has no impact on *ZLight*, *Quorum* and *Chain* as described earlier in this appendix, with a single exception, related to client extraction of abort histories from the received ABORT messages (see Step P3, Sec. B.5). Namely, if the client receives a history from some replica r_j consisting of a checkpointed state followed by CHK requests, the client will first collapse all such histories into the single checkpointed state (i.e., the client will perform the checkpoint on behalf of the replica). Only in case the client cannot retrieve $t + 1$ confirmations of (some) checkpointed state when executing Step P3 in this way, the client will repeat the procedure described in this step with replica histories as received from replicas, i.e., precisely as described in Step P3, Section B.5.

Appendix C

Abstract: correctness proofs

In this Section, we give the correctness proofs of *ZLight*, *Quorum*, and *Chain*. Moreover, we omit the correctness proof of *Backup* which is straightforward due to the properties of the underlying BFT. Finally, the proofs of liveness of our compositions trivially rely on the assumption of an exponentially increasing *Backup* configuration parameter k (Sec. 3.4.3). Since *ZLight* and *Quorum* share many similarities, we give their correctness proof together. This is followed by the proof of *Chain*.

C.1 ZLight and Quorum

In this Section, we prove that *ZLight* and *Quorum* implements Abstract. We first prove the common properties of the two implementations and then focus on the only different property (Progress).

Well-formed commit indications. It is easy to see that the reply returned by a commit indication always equals $rep(h)$), where (commit history) h is a uniquely defined sequence of requests. Indeed, by Step Z4/Q3, in order to commit a request a client needs to receive identical digests of some history h' and identical reply digests from all $3f + 1$ replicas including at least $2f + 1$ correct ones. By Step Z3 of *ZLight* (reps., Q2 of *Quorum*), a digest of the reply sent by a correct replica is $D(rep(h')$. Hence, h' is exactly a commit history h and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica executes an invoked request before sending a RESP message in Step Z3 (resp., Q2), it is straightforward to see that if req is committed with a commit history h_{req}, then req is in h_{req}.

Validity. For any request req to appear in a commit or abort history, at least $f + 1$ replicas must have sent a history (or a digest of a history) containing req to the client (see Step Z4/Q3 for commit histories, and Step P3 for abort histories). Hence, at least one correct replica appended req to its local history. By Step Z3/Q2, the correct replica r_j appends req to its local history only if r_j receives a REQ message with a valid MAC from a client. This is, in turn, present only if some client invoked req, or if req is contained in some verified (valid) init history.

Moreover, by Step Z3/Q2, no replica executes the same request twice (since every replica maintains $t_j[c]$). Hence, no request appears twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step P3 Sec. B.5).

Termination. By assumption of a quorum of $2f + 1$ correct replicas and fair-loss links: (1) correct replicas eventually receive the PANIC message sent by correct client c (in Step P1) and (2) c eventually receives $2f + 1$ abort messages from correct replicas (sent in Step P2). Hence, if correct client c panics, it eventually aborts invoked request req, in case c does not commit req beforehand.

To prove Commit and Abort Ordering we first prove the following Lemma.

Lemma 1 *Let r_j be a correct replica and LH_j^{req} the state of LH_j upon r_j executes req. Then, LH_j^{req} remains a prefix of LH_j forever.*

A correct replica r_j modifies its local history LH_j only in Step Z3/Q2 by sequentially appending requests to LH_j. Hence, LH_j^{req} remains a prefix of LH_j forever.

Commit Order. Assume, by contradiction, that there are two committed requests req (by benign client c) and $req' \neq req$ (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. Since a benign client commits a request only if it receives in Step Z4/Q3 identical digests of replicas' local histories from all $3f + 1$ replicas, there must be a correct replica r_j that sent $D(h_{req})$ to c and $D(h_{req'})$ to c' such that $h(req)$ is not a prefix of $h_{req'}$ nor vice versa. A contradiction with Lemma 1.

Abort Order. First, we show that for every committed request req with the commit history h_{req} and any ABORT message m sent by a correct replica r_j containing local history LH_j^m, h_{req} is a prefix of LH_j^m. Assume, by contradiction, that there are request req', correct replica $r_{j'}$ and ABORT message m' such that the above does not hold. Then, since a benign client needs to receive identical history digests from all replicas to commit a request (Step Z4/Q3), and since $r_{j'}$ stops executing new requests before sending any ABORT message (Step P2), $r_{j'}$ executed req before sending m'. However, by Lemma 1, $h_{req'}$ is a prefix of $LH_{j'}^{m'}$ — a contradiction.

By Step P3, a client that aborts a request waits for $2f + 1$ ABORT messages including at least $f + 1$ from correct replicas. Since any commit history h_{req} is a prefix of every history sent in an ABORT message by any correct replica (as shown above), at least $f + 1$ received histories will contain h_{req} as a prefix, for any committed request req. Hence, by construction of abort histories (Step P3 Sec. B.5) every commit history h_{req} is a prefix of every abort history.

Init Order. By the clarifications of Step Z3/Q2 and Step P2 given in Section B.6, every correct replica must initialize its local history (with some valid init history) before sending any RESP or ABORT message. Since any common prefix CP of all valid init histories is a prefix of any particular init history I, CP is a prefix of every local history sent by a correct replica in an RESP or ABORT message. Init Order for commit histories immediately follows. In the case of abort histories, notice that at least out of $2f + 1$ ABORT messages received by a client on aborting a request in Step P3, at least $f + 1$ are sent by correct processes and contain local histories that have CP as a prefix. Hence, by Step P3, CP is a prefix of any abort history.

ZLight Progress. Recall that *ZLight* guarantees to commit clients' requests if: there are no replica/link failures and Byzantine client failures. We assume that the message processing at processes takes negligible time and that clients set the timer T triggered in Step Z1 to 3Δ. Then, to prove Progress, we prove a stronger property that no client executes Step P1 and panics (consequently no client ever aborts and Progress follows from Termination).

Assume by contradiction that there is a client c that panics and denote the first such time by t_{PANIC}. Since no client is Byzantine, c must be benign and c invoked request req at

$t = t_{PANIC} - 3\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that: (i) by $t + \Delta$ the primary receives req and take Step Z2 and (ii) by time $t + 2\Delta < t_{PANIC}$ the replicas take Step Z3 for req. Since the primary is correct all replicas execute all requests received before t_{PANIC} in the same order (established by the sequence numbers assigned by the primary). Hence, by $t + 3\Delta = t_{PANIC}$, c receives $3f + 1$ identical replies (Step Z4), commits req and never panics. A contradiction.

Quorum Progress. Recall that *Quorum* guarantees to commit clients' requests only if:

- there are no replica/link failures,
- no client is Byzantine, and
- there is no contention.

We assume that the message processing at processes takes negligible time and that the timer T triggered in Step Z1 to 2Δ. Like in the proof of ZLight Progress, we prove a stronger property that no client executes Step P1 and panics.

Assume by contradiction that there is a client c that panics and denote the first such time by t_{PANIC}. Since no client is Byzantine, c must be benign and c invoked request req at $t = t_{PANIC} - 2\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that by time $t + \Delta < t_{PANIC}$ all replicas receive req and take Step Q2. Since there is no contention and all replicas are correct, all replicas order all requests in the same way and send identical histories to the clients. Hence, by $t + 2\Delta = t_{PANIC}$, c receives $3f + 1$ identical replies (Step Q3), commits req and never panics. A contradiction.

C.2 Chain

In this Section, we prove that *Chain* implements Abstract with Progress equivalent to that of *ZLight*. We denote the *predecessor (resp., successor) set* of the replica r_j, by $\overleftarrow{R_j}$ (resp., $\overrightarrow{R_j}$). We also denote by Σ_{last} the set of the last $f + 1$ replicas in the chain order, i.e., $\Sigma_{last} = \{r_j \in \Sigma : i > 2t\}$. In addition, we say that correct replica r_j executes req *at position* pos if $sn_j = pos$ when r_j executes req.

Before proving Abstract properties, we first prove two auxiliary lemmas. Notice also that Lemma 1, Section C.1, extends to *Chain* as well.

Lemma 2 *If correct replica r_j executes req (at position sn, at time t_1), then all correct replicas s_j, $1 \leq j < i$ execute req (at position sn, before t_1).*

By contradiction, assume the lemma does not hold and fix r_j to be the first correct replica that executes req (at position sn), such that there is a correct replica r_x ($x < j$) that never executes req (at position sn); we say r_j is the first replica for which req skips. Since CHAIN messages are authenticated using CAs, r_j executes req at position sn only if r_j receives a CHAIN message with MACs authenticating req and sn from all replicas from $\overleftarrow{R_j}$ authenticate req and sn, i.e., only *after* all correct replicas from $\overleftarrow{R_j}$ execute req at position sn. If $r_x \in \overleftarrow{R_j}$, r_x must have executed req at position sn — a contradiction. On the other hand, if $r_x \notin \overleftarrow{R_j}$, then r_j is not the first replica for which req skips, since req skips for any correct replica (at least one) from $\overleftarrow{R_j}$ — a contradiction.

Lemma 3 *If benign client c commits req with history h (at time t_1), then all correct replicas in Σ_{last} execute req (before t_1) and the state of their local history upon executing req is h.*

To prove this lemma, notice that correct replica $r_j \in \Sigma_{last}$ generates a MAC for the client authenticating req and $D(h')$ for some history h' (Step C3): (1) only after r_j executes req and (2) only if the state of LH_j upon execution of req equals h'. Moreover, by Step C3, no correct replica executes the same request twice. By Step C4, a benign client (resp., replica) cannot commit req with h unless it receives a MAC authenticating req and $D(h')$ from every correct replica in Σ_{last}. Hence the lemma.

Well-formed commit indications. By Step C4, in order to commit a request a client needs to receive MACs authenticating $LHDigest = D(h')$ for some history h' and the reply digest from all replicas from Σ_{last}, including at least one correct replica. By Step C3, a digest of the reply sent by a correct replica is $D(rep(h')$. Hence, h' is exactly a commit history h and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica executes an invoked request before sending a CHAIN message in Step C3, it is straightforward to see that if req is committed with a commit history h_{req}, then req is in h_{req}.

Validity. For any request req to appear in a abort (resp., commit) history h, at least $f + 1$ replicas must have have sent h (resp., a digest of h) in Step P2 (resp., Step C3) such that $req \in h$. Hence, at least one correct replica executed req.

Now, we show that correct replicas execute only requests invoked by clients. By contradiction, assume that some correct replica executed a request not invoked by any client and let r_j be the first correct replica to execute such a request req' in Step C3 of *Chain*. In case $j < f+1$, r_j executes req' only if r_j receives a CHAIN message with a MAC from the client, i.e., only if some client invoked req, or if req is contained in some valid init history. On the other hand, if $j > f + 1$, Lemma 2 yields a contradiction with our assumption that r_j is the first correct replica to execute req'.

Moreover, by Step C3, no replica executes the same request twice (every replica maintains $t_j[c]$). Hence, no request appears twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step P3(ii) Sec. B.5).

Termination. See the proof of Termination for *ZLight*/*Quorum* (Sec. C.1).

Moreover, to see that a committed request req must be in its commit h_{req}, notice that a client needs to receive the MAC for the same local history digest $D(h_{req})$ from all $f+1$ from Σ_{last} including at least one correct replica r_j. By Step C3, r_j executes req and appends it to its local history LH_j before authenticating the digest of LH_j; hence, $req \in h_{req}$.

Commit Order. Assume, by contradiction, that there are two committed request req (by benign client c) and $req' \neq req$ (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. By Lemma 3, there is correct replica $r_j \in \Sigma_{last}$ that executed req and req' such that the state of LH_j upon executing these requests is h_{req} and $h_{req'}$, respectively. A contradiction with Lemma 1 (recall that this lemma extends to *Chain* as well).

Abort Order. Assume, by contradiction, that there is committed request req_C (by some benign client) with commit history h_{req_C} and aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. By Lemma 3 and the assumption of at most f faulty replicas, all correct replicas (at least one) from Σ_{last} execute req_C and their state upon executing req_C is h_{req_C}. Let $r_j \in \Sigma_{last}$ be a correct replica with the highest index ind among all replicas in Σ_{last}. By Lemma 2, all correct replicas execute all the requests in h_{req_C} at the same positions these requests have in h_{req_C}. In addition, a correct replica executes all the requests belonging to h_{req_C} before sending any ABORT message in

Step P2; indeed, before sending any ABORT message, a correct replica must stop further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an ABORT message, h_{reqC} is a prefix of LH_j.

Finally, by Step P3, a client that aborts a request waits for $2f + 1$ ABORT messages including at least $f + 1$ from correct replicas. By construction of abort histories (Step P3) every commit history, including h_{reqC} is a prefix of every abort history, including h_{reqA}, a contradiction.

Init Order. The proof is identical to the proof of *ZLight*/*Quorum* Init Order.

Progress. *Chain* guarantees to commit clients' requests under the same conditions as *ZLight*, i.e., if: there are no replica/link failures and Byzantine client failures. Assuming that the message processing at processes takes negligible time, it is sufficient that clients set the timer T triggered in Step C1 to $(3f + 2)\Delta$. Then, Progress of *Chain* is very simple to show, along the lines of *ZLight* Progress (Sec. C.1).